AD-A104 160 NAVAL SURFACE WEAPONS CENTER DAHLGREN VA F/G 9/2
SIMPL-Q REFERENCE MANUAL,(U)
MAY 81 C J NAPLES
UNCLASSIFIED NSWC-TR-81-262 NL

1 OF 2
AD
A104 160

AD A104160

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER NSWC TR-81-262	2. GOVT ACCESSION NO. AD-A104160	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) SIMPL-Q REFERENCE MANUAL		5. TYPE OF REPORT & PERIOD COVERED
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Charles J. Naples		8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION NAME AND ADDRESS Naval Surface Weapons Center (K105) Dahlgren, VA 22448		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS NIF
11. CONTROLLING OFFICE NAME AND ADDRESS Naval Surface Weapons Center (K105) Dahlgren, VA 22448		12. REPORT DATE May 1981
		13. NUMBER OF PAGES 164
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)		15. SECURITY CLASS. (of this report) UNCLASSIFIED
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Computer programming language
System programming language
macro language
intermediate language machine
SIMPL programming language

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

The SIMPL-Q system ~~described~~ in this report consists of a system computer programming language (SIMPL-Q) and a simple macro language (SML). The SIMPL-Q compiler and SML execute on either the CDC-6700 under the SCOPE 3.4 operating system or on the Nanodata QM-1 under the EASY-II operating system. The code produced by the SIMPL-Q system is designed to execute on the Nanodata QM-1 EASY intermediate language machine.

continued

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

20 (continued)

The SIMPL-Q language is a member of the SIMPL-T family of languages. SIMPL-T was designed and implemented on the UNIVAC 1108 by ~~Dr.~~ Victor R. Basili and Albert J. Turner, ~~of the~~ University of Maryland. SIMPL-T is a transportable structured programming language with three basic data types: integer, character, and string. SIMPL-T was enhanced to allow it to be used as a system programming language for the Nanodata QM-1, hence the name SIMPL-Q.

This report, which is intended to be used as a reference manual, and supersedes TR-3778. The report presents the details, rules, and examples for writing programs in the SIMPL-Q language. It also contains sufficient information to prepare and compile such programs.

FOREWORD

This report replaces TR-3778 as a reference manual for the computer programming language SIMPL-Q. SIMPL-Q is an elementary structured system programming language for the Nanodata QM-1 computer. It is a member of the SIMPL family of languages that are designed to be relatively machine-independent and whose compilers are relatively transportable onto a variety of machines. SIMPL-Q is based on SIMPL-T, the transportable member of the SIMPL family. The first member of the SIMPL family, the typeless language SIMPL-X, was bootstrapped onto the Univac 1108 computer in the Fall of 1972. The implementation of SIMPL-T was completed in January 1974. SIMPL-T was bootstrapped onto the CDC 6700 computer in the Fall of 1975. SIMPL-T was enhanced to be used as the system programming language for the Nanodata microprogrammable QM-1 computer.

Dr. Victor R. Basili, associate professor of Computer Science at the University of Maryland, was the initial primary designer of the SIMPL (SIMPL-X, SIMPL-T) family of languages. Mr. Albert J. Turner, candidate for a Doctor of Philosophy in Computer Science at the University of Maryland, was the primary implementor for the Univac 1108 computer version. The SIMPL macro language was designed by Mr. Jack A. Verson and Mr. Robert E. Noonan at the University of Maryland. Mr. John G. Perry, Jr. bootstrapped the Univac 1108 computer SIMPL-T version to the CDC 6700 computer. Mr. Charles W. Flink, II, and Mr. Perry designed the SIMPL-Q enhancements. Mr. Perry was the implementor for the SIMPL-Q version. Mr. Alan J. Hynson designed and implemented the micro-code support routines for the Nanodata QM-1. Mr. Charles Naples designed and implemented the INPUT/OUTPUT routines and Mr. Jay C. Meyers implemented the extended precision procedures. Mrs. Anne Ammerman of COMPRO assisted in the preparation of the original reference manual (TR-3778). Mrs. Carolyn Tilghman and Mr. Don Oates of SDC assisted in the preparation of this document.

This report was prepared in the Programming Systems Branch of the Computer Division. The project was performed to support the TRIDENT emulator high-level language debug system. This project was funded by the Computer Facilities Division and the FBM Geoballistics Division. The report was reviewed by Robert Bevan, Head, Programming Systems Branch.

Released by:

[illegible]

ROBERT T. RYLAND, JR., Head
Strategic Systems Department

TABLE OF CONTENTS

TABLE OF CONTENTS (Cont'd)

TABLE OF CONTENTS (Cont'd)

TABLE OF CONTENTS (Cont'd)

TABLE OF CONTENTS (Cont'd)

1. INTRODUCTION

This manual describes the language SIMPL-Q which is used in conjunction with the EASY II operating system for higher-level programming on the Nanodata QM-1 computer. The reader may consider this document as a replacement for TR-3778 which describes an earlier version of SIMPL-Q which ran under the original EASY operating system. SIMPL-Q is essentially an extension of the language SIMPL-T, which is described in: V. R. Basili, "SIMPL-T, A Structured Programming Language," Computer Note CN-14.2, University of Maryland Computer Science Center, August 1975. Much of this report is taken directly from the manual mentioned above.

SIMPL-Q is a procedure-oriented, non-block-structured programming language that was designed to conform to the standards of structured programming and modular design. SIMPL-Q is intended to be a systems programming language for the Nanodata QM-1 machine. SIMPL-Q is a member of the SIMPL family of languages. The initial design and development of the SIMPL family of languages was done at the University of Maryland Computer Science Center and Computer Science Department.

In order to avoid the inclusion of much material that would be superfluous for the anticipated readers, it is assumed that the reader has some knowledge of a general-purpose programming language such as FORTRAN, BASIC, ALGOL, or PL/I.

The manual is designed so that the basic features are presented first, and more specialized features are presented later. Section 2 contains a description of the basic language that is sufficient to get a novice SIMPL-Q programmer "on the air". Section 2, Section 3, and most, if not all, of Section 4 contain the material that would normally be covered in a programming course (with selected topics from Section 5 perhaps also being included). Most of the material in Section 5 is for those who are familiar with the language and have special purpose requirements. Section 6 contains information about some enhancements to the basic SIMPL-Q language to support the Nanodata QM-1. Section 7 contains information about using the SIMPL-Q compiler, and Section 8 contains assorted information about SIMPL-Q implementation.

A language feature that has not yet been explained is sometimes used in an example in order to provide a more illustrative example. The meaning of such a feature should be clear from its usage to those who are familiar with another general-purpose programming language but if not, the feature is always explained soon after such a usage. It should also be noted that the examples are designed <u>primarily</u> to illustrate the SIMPL-Q language and, thus, they may not always illustrate the best way to solve a particular programming problem.

Braces ({ }) are used to denote optional syntax and the symbols < and > are used to enclose the name of a general syntactic entity. For example, the syntax for a call statement can be specified by

```
CALL <identifier> {(<parameter list>)}
```

This means that a call statement consists of the word CALL followed by an identifier. The identifier may optionally be followed by a parameter list, enclosed in parentheses. Words such as CALL are called <u>keywords</u>.

2. THE BASIC SIMPL-Q LANGUAGE

2.1 PROGRAM STRUCTURE

The syntax for a SIMPL-Q program is illustrated by

<module id> {<declaration list>} <segment list> START

The <module id> assigns a name to the compile module (see Section 3.15 for format). The <declaration list> defines the variables that may be used anywhere in the program. The <segment list> is a collection of procedures (subroutines) and functions. (The <segment list> may consist of only a single procedure.)

The following example illustrates this program structure.

```
MODULE STRING TEST [20] = 'TEST PROGRAM'     }   module id
INT X,Y                                      }   declaration list

ENTRY PROC PRINTSUM (INT A, INT B)           \
WRITE (A+B)                                  |
                                             |
PROC MAINPROG                                |
X := 3                                       >   segment list
Y := 4                                       |
CALL PRINTSUM (X,Y)                          |
START                                        /
```

(The result of this program is that 7 is printed.)

Thus, a SIMPL-Q program contains a (possibly empty) set of global declarations and a set of procedures and functions. Execution begins with one of the procedures, and the procedures and functions are called as needed during execution.

2.1.1 Declarations

The initial declaration list of a program contains declarations for all variable identifier names that are global. A global identifier is an identifier that is known to all segments of a program. Only global declarations for integer variables and integer arrays are discussed in this section.

2.1.1.1 Integer Declaration

An integer variable may have any integer value between $-2^{35} + 1$ and $2^{35} - 1$, inclusive. An integer variable declaration consists of the keyword INT followed by one or more identifier names, separated by commas. Initialization may also be specified as illustrated by the following valid declaration list.

```
INT X

INT CAT, DOG1

INT M=3, N=-1, I
```

In the above example, M and N are initialized to the values 3 and -1, respectively. This means that these variables will have the specified values when execution of the program begins. The value of an uninitialized variable is initially undefined. (Actually, globals will have value 0 on the QM-1.)

2.1.1.2 Integer Array Declaration

The only data structure in SIMPL-Q is the one-dimensional array. This is an ordered collection of elements, all of the same data type. The elements are numbered 0, 1,..., n-1, where n is the number of elements in the array.

Integer array declarations begin with the keywords INT ARRAY, and are completed by listing the array identifiers and the number of elements for each array. The number of elements must be a positive integer, and is enclosed in parentheses. For example,

```
INT ARRAY TOTALS(10)
```

declares an array of 10 elements: TOTALS(0), TOTALS(1),..., TOTALS(9).

An array can also be initialized by specifying a list of values for the array elements. Initialization begins with the first element (number 0) and proceeds until the list is exhausted (or all array elements are exhausted). A repetition factor can be specified by enclosing the factor in parentheses following the initialization value.

Some examples are:

```
INT ARRAY A(3), BAT(95), VECTOR(20)

INT ARRAY A1(10), B(5) = (2,3,-1)

INT ARRAY C(11) = (0,1,3(9))
```

The second declaration specifies that B(0), B(1), and B(2) are to be initialized to 2, 3, and -1, respectively. The third declaration initializes C(0) to 0, C(1) to 1, and C(2)-C(10) to 3.

2.1.1.3 Declaration List

A declaration list, such as the list of global declarations at the beginning of a program, consists of one or more declarations. Declarations follow one another with no separator (except blanks). More than one declaration for the same type can appear in a declaration list. All identifiers used in a program must be declared.

An example of a declaration list is:

```
INT X, Y

INT I

INT ARRAY INPUTS (100), OUTPUTS(50)

INT SUM

INT ARRAY SUMS(20) = (0(20))
```

2.1.2 Segments

A segment is a procedure or function definition. Segments contain a list of statements to be executed when the segment is invoked.

2.1.2.1 Procedures

The syntax for a procedure definition is illustrated by

```
PROC <identifier> {(<parameter list>)} {<local declaration list>}
     <statement list> {RETURN}
```

where <identifier> is the name of the procedure.

An example of a procedure definition is:

```
PROC TEST (INT X, INT Y)
/* THIS PROC PRINTS THE SUM OF X AND Y */
WRITE (X+Y)
```

A procedure is a subroutine that, when invoked, executes its <statement list> and returns to the caller. A procedure may access any global identifier (unless the procedure has a local identifier by the same name) as well as its local identifiers and parameters.

The items of the <parameter list>, separated by commas, are of the form INT <identifier> or INT ARRAY <identifier>. These parameters are passed to the procedure when it is invoked (called).

Integer parameters are passed by value (unless otherwise specified in Section 4.2). This means that if a procedure changes the value of an integer parameter, the new value is effective only to that procedure. For example, if procedure P is defined by

```
PROC P(INT X)
X := 7
```

and the statements

```
X := 3
CALL P(X)
WRITE(X)
```

are executed, then the number printed will be 3 (not 7).

Array parameters, however, are passed by reference. Logically, this means that the array itself is passed (rather than the value as for integer parameters). Thus, any modification to an array parameter by a procedure will be a modification to the actual array passed as an argument by the caller. For example, if procedure Q is defined by

```
PROC Q(INT I, INT ARRAY A)
A(I) := 7
```

and the statements

```
A(2) := 3
CALL Q(2, A)
WRITE(A(2))
```

are executed, then 7 will be printed.

2.1.2.2 Functions

The function definition syntax is illustrated by

```
INT FUNC <identifier> {(<paramater list>)} {<local declaration list>}
     {<statement list>} RETURN(<expression>)
```

A function is similar to a procedure. The main differences are:

1. The value of <expression> is returned (as the value of the function evaluation) to be used in the same manner as the value of a variable would be used.

2. Functions may not have side effects; that is, they may not change the values of any nonlocal variables or arrays.

(Note that item 2 is assumed but not enforced. Those who insist on writing functions with side effects should see Section 8.4.)

2.1.2.3 Local Declarations

All local variables and arrays must be declared in the local declaration list. Local declarations are similar to global declarations, but initialization is not allowed. (The values of local variables at entry to a segment are undefined.) A maximum of 8191 QM-1 words is allowed for local variables and compiler temporaries.

2.1.3 Scope of Identifiers

Global identifiers, including segment names, are accessible from all segments unless a segment declares a local with the same name as a global. Local declarations override global declarations so that a global identifier is not available to a segment in which that identifier is declared local.

Local identifiers are only accessible to the segment in which they are declared. Both globals and locals may be passed as parameters. The value of all locals is undefined at entry to the segment, and locals do not necessarily retain their values between successive calls to the segment.

2.2 COMMENTS AND BLANKS

Blanks may appear anywhere in a SIMPL-Q program except within an identifier, symbol, keyword, or constant. Blanks are significant delimiters and may be needed as separators for identifiers or constants. For example,

```
IF X
```

and

```
IFX
```

are not equivalent.

A comment is any character string enclosed by /* and */. (See Section 6.1 for a modification of this convention.) A comment may appear anywhere that a blank may occur and has no effect on the execution of a program. The following illustrates a comment:

```
/* THIS IS A COMMENT. */
```

2.3 STATEMENTS

The syntactic entity <statement list> denotes any sequence of SIMPL-Q statements. No separators (other than blanks) are used between statements.

2.3.1 Assignment Statement

The syntax of the assignment statement is given by

```
<variable> := <expression>
```

where <variable> is either a simple variable (i.e., an integer identifier) or a subscripted variable. The assignment statement causes the value of the <expression> to be assigned to the <variable>. Examples of valid SIMPL-Q assignment statements are:

```
X := Y+Z

X := Y=Z

A(I):= A(I+1)+A(J-2)*X
```

2.3.2 IF Statement

The IF statement causes conditional execution of a sequence of one or more statements. The syntax is

```
IF <expression>
     THEN <statement list>1
     {ELSE <statement list>2} END
```

At execution, the value of the <expression> determines the action taken. If the value is nonzero, <statement list>$_1$ is executed and <statement list>$_2$ (if there is an else part) is skipped. If the value is zero, <statement list>$_2$ (if it exists) is executed and <statement list>$_1$ is not executed. Execution proceeds with the next statement (following END) after execution of either <statement list>.

Example

```
IF X<3 .AND. Y<X
     THEN
          Y:=X
     ELSE
          X:=X+1
          Y:=Y-1
          IF X>Y
               THEN
                    X:=Y
               END
     END
```

Note that the ELSE part of the main IF statement also contains an IF statement that will be executed only if the ELSE part is executed.

Example

```
IF X THEN Y:=Y/X ELSE Y:=Y/2  END
```

This statement divides Y by X if X is nonzero and divides Y by 2 if X is zero.

2.3.3 While Statement

The WHILE statement provides a means of iteration (looping):

```
WHILE <expression> DO <statement list> END
```

The value of the <expression> determines the action at execution time, just as for the IF statement. If the value of <expression> is nonzero, then <statement list> is executed; otherwise, <statement list> is skipped and execution proceeds with the statement following END. However, if <statement list> is executed, then execution proceeds with the WHILE statement again. Thus, if <expression> is nonzero, then <statement list> is executed until <expression> becomes zero.

Example. The following statement list sums the odd and even integers from 1 to 100.

```
ODD := 0
EVEN := 0
I := 0
WHILE I<100
DO
     I := I+1
     IF I/2 * 2 = I
          THEN/* EVEN INTEGER */
               EVEN := EVEN + I
          ELSE /* ODD */
               ODD : = ODD + I
          END
END
```

2.3.4 Case Statement

Exactly one of a group of statement lists may be executed by using the CASE statement. The syntax is illustrated by

```
CASE <expression> OF
    \n₁\<statement list>₁
    \n₂\<statement list>₂
              .
              .
              .
    \nK\<statement list>K
   {ELSE <statement list>K+1} END
```

where each $n_1, n_2, \ldots, n_K$ is a constant between 0 and 255.

If the value of <expression> is n_j, then <statement list>$_j$ is executed and the other statement lists are not executed. If <expression> does not evaluate to any of the n_i's, then the ELSE (<statement list>$_{K+1}$) is executed, if there is an ELSE, and none of the statement lists are executed if there is no ELSE. Defined macros may be used in case designators (\<MACRO>\). The cases may be in any order, and more than one case designator \n_i\ may be used with the same statement list, as illustrated in the following example.

Example

```
CASE X*Y+Z OF
    \1\
        X := 3
   \2\
        If X<Y
            THEN
                X := Y
            END
        Y := Y+1
  \4\\6\/* CASES 4 AND 6 COINCIDE */
        X := 2
        Y := 3
   ELSE
        X := 0
   END
```

2.3.5 Call Statement

The CALL statement

```
CALL <identifier> {(<argument list>)}
```

causes the procedure named <identifier> to be executed. Each argument in the argument list may be an expression or an array, and the arguments must agree in number and type with the parameters in the procedure definition for the procedure that is called. Arguments in <argument list> are separated by commas. The <argument list> may have a maximum of 22 parameters.

Upon completion of the execution of the procedure, execution resumes with the statement following the CALL statement.

Example. To invoke the procedure DOIT with arguments X+Y and the array A, the statement

```
CALL DOIT (X+Y, A)
```

is used.

2.3.6 Example

```
PROC SORT (INT N, INT ARRAY A)
/* THIS PROCEDURE USED A BUBBLE SORT ALGORITHM TO SORT THE
   ELEMENTS OF ARRAY 'A' INTO ASCENDING ORDER.  THE VALUE
   OF THE PARAMETER 'N' IS THE NUMBER OF ITEMS TO BE SORTED.  */
INT SORTED, /* SWITCH TO INDICATE WHETHER FINISHED */
    LAST,   /* LAST ELEMENT THAT NEEDS TO BE CHECKED */
    I,      /* FOR GOING THROUGH ARRAY */
    SAVE    /* FOR HOLDING VALUES TEMPORARILY */
IF N>1
  THEN /* SORT NEEDED */
    SORTED := 0 /* INDICATE NOT FINISHED */
    LAST := N-1 /* START WITH WHOLE ARRAY */
    WHILE .NOT. SORTED
     DO /* CHECK CURRENT SEQUENCE FOR CORRECTNESS */
       SORTED := 1 /* ASSUME FINISHED */
       I := 1      /* INITIALIZE ELEMENT POINTER */
       WHILE I <= LAST
         DO /* COMPARE ADJACENT ELEMENTS UP TO 'LAST' */
           IF A(I-1) > A(I)
             THEN /* OUT OF ORDER */
               SAVE := A(I)       /* INTERCHANGE */
               A(I) := A(I-1)     /* A(I) AND    */
               A(I-1) := SAVE     /* A(I-1)      */
               SORTED := 0        /* MAY NOT BE FINISHED */
             END
           I := I+1
         END /* LOOP FOR COMPARING ADJACENT ELEMENTS */
       /* A(LAST),..., A(N-1) ARE NOW OK */
       LAST := LAST -1
     END /* LOOP FOR CHECKING CURRENT SEQUENCE */
 END /* IF N>1 */
/* END PROC 'SORT' */
```

2.4 INTEGER EXPRESSIONS

An integer expression represents an integer value. An integer expression may be:

1. A scalar integer variable (either a simple variable or a subscripted array variable)

2. An integer constant

3. An integer function call

4. An integer operation (such as + or -) where each operand may also be an expression

5. An integer expression enclosed in parentheses

2.4.1 Subscripted Array Variables

An array element is designated by following the array name with a subscript, enclosed in parentheses, whose value designates the number of the array element to be used. The subscript can be an integer expression.

For example

```
A(3)
```

designates the 4th element of array A, while

```
A(X + A(Y))
```

designates the element whose number is the value of X plus the value of the array element designated by A(Y).

2.4.2 Function Calls

A function call has the form

```
<identifier> {(<argument list>)}
```

where <identifier> is the name of the function. The rules for <argument list> are the same as for the CALL statement.

2.4.3 Constants

An integer constant may be designated by any sequence of decimal digits representing a valid non-negative integer value. Note that negative constants may usually be used where desired although such a constant is formally viewed as the unary minus operation on a non-negative constant integer expression.

For example, the following are valid SIMPL-Q integer constants.

```
3

35927

0

123456789
```

2.5 BASIC INTEGER OPERATORS

The operators described in this section all have integer expressions as operands and yield an integer result. Any arithmethic overflow that occurs in a calculation is ignored.

2.5.1 Arithmetic Operators

Addition (+), subtraction (-), and multiplication (*) are binary operators with the usual meaning. The integer divide (/) operator yields the integer quotient of its operands. Thus, if the result of X/Y is Q, then X = Q*Y + R, where R is the remainder that was discarded in the integer divide.

The unary minus (-) operator yields the negative of its operand. Note that the expression -3 is formally viewed as the unary minus operation on the constant 3 although it would probably be logically (and equivalently) viewed as the constant "minus three" by the programmer. There is no unary plus operator in SIMPL-Q.

2.5.2 Relational Operators

The relational operators are equal (=), not equal (<>), less than (<), less than or equal (<=), greater than (>), and greater than or equal (>=). The expression X=Y has value 1 if X and Y are equal, and value 0 otherwise. The remaining relational operators are similarly defined.

Note that the result of a relational operation always has value 1 or zero, depending on whether the relation is true or false, respectively. The relational operators can also be denoted by .EQ., .NE., .LT., .LE., .GT., and .GE., respectively.

2.5.3 Logical Operators

The logical operators .AND., .OR., and .NOT. are defined by:

X.AND.Y is 1 if both X and Y are nonzero, and is 0 otherwise

X.OR.Y is 0 if both X and Y are zero, and is 1 otherwise

.NOT.X is 1 if X is zero and is 0 otherwise

As is the case for relational operations, a logical operation always yields the result 1 or 0.

Note that the logical operators yield the "natural" result. For example, the expression

```
X<Y .AND. Y<Z
```

will have the value 1 (i.e., will be "true") if Y is both greater than X and less than Z, and will have the value 0 (i.e., will be "false") otherwise.

2.5.4 Precedence

The precedence of the basic integer operations, from highest to lowest, is

.NOT. - (unary)	unary
* /	arithmetic
+ - (binary)	
= <> < > <= >=	relational
.AND.	logical
.OR.	

The order of evaluation between operators of equal precedence is left to right (except between unary operators, which is right to left).

As an example, the expression

```
- A + B + C * D
```

would be evaluated by

1. Negating the value of A
2. Multiplying the value of C by the value of D
3. Adding the value of B to the result from (1)
4. Adding the results from (2) and (3)

Parentheses may be used to alter the normal precedence. Thus, (A+B)*C would cause the values of A and B to be added and the result to be multiplied by the value of C.

2.5.5 Examples

The following are examples of valid SIMPL-Q expressions.

1. X + Y/7 * 2
2. X<3 .OR. X>8
3. X>3 .AND. X+Y<10
4. X + (X*(Y+1)<500)

For X=9 and Y=12, these expressions have the values

1. 11
2. 1
3. 0
4. 10

2.6 IDENTIFIERS

Identifiers (i.e., names) in SIMPL-Q may be any string of letters or digits that begins with a letter. For usage in an identifier, the symbol $ is considered to be a letter. Identifiers are used to denote variables, arrays, procedures, functions, and other entities in a program. All identifiers used in a program (except SIMPL-Q intrinsic identifiers, such as READ) must be declared.

There is no formal restriction on the length of identifiers. However, identifiers may not cross the boundary of a source input record (e.g., card), so that there is an actual restriction to the length of an input record (e.g., 80 characters).

Certain reserved words (keywords) may not be used as identifiers in a SIMPL-Q program. These keywords (such as IF, INT) are listed in Appendix V. Due to the special meaning given to these keywords, rather disastrous results may occur if a keyword is used an an identifier in a SIMPL-Q program. This is especially true of keywords used in declarations (such as INT, ARRAY, PROC). The resulting diagnostics generated by the compiler may not be too helpful for such an error, primarily because the programmer often overlooks this type of error as a possible cause of the diagnostics.

Since many keywords are used for more specialized features of the SIMPL-Q language, the list in Appendix V should be consulted before writing a SIMPL-Q program.

2.7 BASIC I/O

2.7.1 READ (not implemented at this time)

READ may be used to read values from job stream input (card, teletype, etc.) into integer variables. Values to be read are placed on input records (cards, teletype lines, etc.) as decimal constants separated by blanks or commas (or both). Negative values are indicated by placing a minus sign before the number to be read. A value may not cross the boundary of an input record.

To illustrate the READ statement, the statements

```
READ(X,Y,A(I+J))
READ(I,J)
```

and the input

```
3, -2, 5 7
10, 12
```

would cause the same results as

```
X := 3
Y := -2
A(I+J) := 5
I := 7
J := 10
```

Thus, the input to be read in is considered to be a stream of numbers rather than a sequence of cards (or lines, etc.). The numbers are read one by one and numbers are not skipped unless explicit directions to do so are specified. Skipping to the beginning of an input record can be specified by using the arguments SKIP, SKIP0, SKIP1, SKIP2,..., SKIP9. (SKIP is the same as SKIP1.)

The effect of a skip argument is as follows:

SKIP0 - skip to the beginning of the current card (for reread)

SKIP, SKIP1 - skip to the beginning of the next card, etc.

The skip directive is relative to the last value read from the input stream. Thus, for example, successive

```
READ(X, SKIP)
```

statements would cause the first value to be read from each card that has a value on it, regardless of the number of values on a card.

To illustrate further, the statements

```
READ(X, SKIP, Y)
READ(SKIP, Z)
READ(SKIP)
READ(I)
```

and the input

```
3, 5, 7
2
0
1, 4
10
```

would be the same as the assigments

```
X := 3
Y := 2
Z := 0
I := 1
```

The READ statement can also be used to read in values for an entire array. For example, if X is an integer variable and A is an integer array of 10 elements, the statement

```
READ(X, A)
```

would read the next input item into X, and the following 10 items into A(0), A(1),..., A(9).

The intrinsic function EOI may be used to determine the end of the input. The result of the function EOI is given by

$$\text{EOI} = \begin{cases} 1 \text{ if no more items are available for READ} \\ 0 \text{ if one or more items remain to be read} \end{cases}$$

Note that the value of EOI is determined on the basis of values, not input records (such as cards), remaining to be read. The use of EOI is illustrated in Section 2.7.3.

2.7.2 WRITE

Values of expressions may be printed by using WRITE. The values to be printed are considered to be a stream of values that are placed at tab positions that provide columns eight characters in width. A line is not printed until it is filled, unless a skip or eject argument is used.

The carriage control parameters that can be used are:

EJECT - skip to the top of the next page

SKIP0 - start over on the current line (overprint)

SKIP, SKIP1 - print the current line

SKIP2 - print the current line and double-space

$$\left.\begin{array}{l}\text{SKIP3}\\ \quad.\\ \quad.\\ \quad.\\ \text{SKIP9}\end{array}\right\}\ \text{similar to SKIP2}$$

Each argument of WRITE may be an expression, an array, or a carriage control specification. As an example, if X = 3, Y = 2, and I = 10, the statements

```
WRITE(X, 2*X + 3*Y)
WRITE(I, I/X, SKIP, Y)
WRITE(SKIP)
```

would cause

```
3    12     10     3
2
```

to be printed.

To illustrate the use of WRITE with an array argument, if A is an array with 20 elements, the statement

```
WRITE(A)
```

is equivalent (assuming that I is not used for something else) to

```
I := 0
WHILE I<20
  DO WRITE(A(I))
     I := I+1  END
```

2.7.3 Example

```
MODULE STRING SORT [40] = 'SORTS 100 ITEMS IN ASCENDING ORDER'
/* THIS PROGRAM READS IN A SET OF UP TO 100 NUMBERS, SORTS THEM
   INTO ASCENDING ORDER, AND PRINTS OUT TWO COLUMNS IN WHICH THE
   LEFT COLUMN IS THE ORIGINAL SET OF NUMBERS AND THE RIGHT COLUMN
   IS THE SORTED SET.  THE PROCEDURE 'SORT' FROM 2.3.6 IS USED. */
INT N /* NUMBER OF VALUES TO SORT */
INT ARRAY A(100),  /* INPUT SET */
          B(100)   /* SORTED SET */

PROC READINPUT
N := 0
WHILE .NOT. EOI .AND. N<100
  DO /* PUT NEXT VALUE INTO 'A' AND 'B' */
    READ(A(N))
    B(N) := A(N)
    N := N+1   /* COUNT VALUES */
  END

PROC PRINT
INT I
I := 0
WHILE I<N
  DO /* PRINT LINES OF OUTPUT */
    WRITE (A(I), B(I), SKIP)
    I := I+1
  END

    .
    .
    .  Proc SORT from 2.3.6
    .
ENTRY PROC READSORTANDPRINT
CALL READINPUT
CALL SORT(N,B)
CALL PRINT
START
```

An example of the result of executing this program is:

```
  5     -25
 -2      -2
  3       0
  0       3
 10       5
-25      10
 17      17
```

2.8 EXAMPLE

```
MODULE STRING BSEARCH[20] = 'BINARY SEARCH'
 /* THIS PROGRAM READS A SEQUENCE OF NOT MORE THAN 100 NONZERO
    INTEGERS.  THE INTEGERS MUST BE IN INCREASING ORDER AND MUST BE
    FOLLOWED BY A 0 (UNLESS 100 INTEGERS ARE TO BE READ).  THE LIST OF
    VALUES READ IS PRINTED.  ADDITIONAL VALUES ARE THEN READ AND A 'BINARY
    SEARCH' IS USED TO DETERMINE IF EACH VALUE IS A MEMBER OF THE SEQUENCE
    READ INITIALLY.  EACH VALUE IS PRINTED WITH ITS POSITION IN THE
    SEQUENCE (0 IF NOT IN THE SEQUENCE).  */

    INT FUNC SIGN(INT X)
    /* FUNCTION WHOSE VALUE IS:  2 IF X>0
                                 1 IF X=0
                                 0 IF X>0     */

    IF X>0
      THEN
        RETURN(2)
      ELSE
         RETURN(X=0)
      END
    /* END FUNC 'SIGN' */

    ENTRY PROC SEARCH
    /* MAIN PROCEDURE */
    INT N, /* NUMBER OF VALUES READ */
        FOUND, /* SWITCH TO INDICATE WHETHER VALUE WAS FOUND */
        INDEX, /* POSITION OF VALUE IN SEQUENCE */
        LO,    /* LOWER BOUND OF INTERVAL FOR SEARCH */
        HI,    /* UPPER BOUND OF INTERVAL FOR SEARCH */
        KEY    /* VALUE READ TO BE LOOKED FOR */
    INT ARRAY TABLE(101)  /* SEQUENCE */
        N := 0            /* INITIALIZE */
        TABLE (0) := 1    /* FIX UP FOR FIRST READ */
        WHILE N<100 .AND. TABLE (N) <>0
          DO /* READ SEQUENCE */
            N := N+1
            READ (TABLE(N))
            IF TABLE(N)<>0
              THEN
                WRITE(TABLE(N))
              END
          END /* LOOP FOR READING SEQUENCE */
        IF TABLE(N) = 0
          THEN
            N := N-1      /* FIX UP FOR COUNTING THE ZERO */
          END
       WRITE(SKIP)        /* END LINE OF SEQUENCY VALUES */
       WHILE .NOT. EOI
       DO /* READ AND LOOK UP VALUES */
         READ(KEY)
         WRITE(KEY)
         /* INITIALIZE FOR SEARCH */
       FOUND := 0
```

```
          HI := N
          LO := 1          /* INITIAL INTERVAL IS WHOLE ARRAY */
          WHILE LO <= HI .AND. .NOT. FOUND
            DO /* BINARY SEARCH */
              INDEX := (LO+HI)/2 /* LOOK AT MIDPOINT OF INTERVAL */
              CASE SIGN(TABLE(INDEX)-KEY) OF
               \1\/* TABLE (INDEX)=KEY -- (FOUND) */
                  FOUND := 1
               \2\/* TABLE (INDEX)>KEY */
                  HI := INDEX-1 /* TRY LOWER INDICES */
               \0\/* TABLE(INDEX)<KEY */
                  LO := INDEX+1/* TRY HIGHER INDICES */
              END
          END /* LOOP FOR BINARY SEARCH */
        IF FOUND
         THEN
           WRITE(INDEX,SKIP)
         ELSE
           WRITE(0,SKIP)
         END
  END /* LOOP FOR READING AND LOOKING UP VALUES */
/* END PROC 'SEARCH' */
START
```

For this program, the input

```
     2    3    5    8    10    11    15    0
     2    1    0    8    7        15    18
```

would produce the output

```
     2    3    5    8    10    11    15
     2    1
     1    0
     0    0
     8    4
     7    0
    15    7
    18    0
```

3. STRING DATA

3.1 INTRODUCTION

In this section a second data type, string, is discussed. A string is a (finite) sequence of characters. The number of characters in the sequence is called the length of the string, and the string of length 0 is called the null string. The characters may be any of the ASCII characters (see Appendix III).

3.2 CONSTANTS

A string constant is denoted by enclosing the string in apostrophes. (Note that computer people usually call apostrophes "quotes".) Any apostrophe in the string is indicated by using two apostrophes. Examples are:

```
'THIS IS A STRING'

'THERE IS AN APOSTROPHE ('') IN THIS STRING'
```

The length of the first string above is 16. The length of the second is 41, and printing it would yield

```
THERE IS AN APOSTROPHE (') IN THIS STRING
```

The null string constant is denoted by ''.

A string constant may not exceed 256 characters in length.

3.3 VARIABLES

A string variable has a maximum length associated with it. The value of a string variable is a string, and the maximum length limits the length of the string value.

3.4 DECLARATIONS

3.4.1 Scalar

A string declaration includes the specification of the maximum length for the value of the string variable. This specification is made by enclosing the maximum length (a positive integer constant) in brackets following the string identifier. The maximum length may not exceed 4095.

Examples are:

```
STRING S[5], T[50]

STRING MESSAGE [10] = 'HELLO'
```

The first declaration defines strings S with maximum length 5, and T with maximum length 50. In the second declaration, the maximum length of MESSAGE is specified to be 10, and MESSAGE is initialized with the value 'HELLO' (so that the current length is initially 5).

The value of an uninitialized string variable is defined.

3.4.2 Array

All elements of a string array must have the same maximum length. Thus, a string array declaration must include the maximum length specification as well as the number of elements (number of strings) in the array.

String array declarations are illustrated by

```
STRING ARRAY INPUT [50] (100)

STRING ARRAY MESSAGES [20] (10) = ('MESSAGE 0', 'MESSAGE 1'),
     SA[13] (25) = ('ABC', 'XYZ' (3), 'CAT')
```

In these declarations, the array INPUT contains 100 strings of maximum length 50 each. MESSAGES (0) is initialized to 'MESSAGE 0' and 'MESSAGE 1' is the initial value of MESSAGES (1). Execution of the statements

```
I := 0
WHILE I<5
  DO
     WRITE (SA(I), SKIP)
     I := I+1
  END
```

at the beginning of the program would produce the output

```
ABC
XYZ
XYZ
XYZ
CAT
```

3.5 OPERATORS

3.5.1 Concatenate

The concatenate operator .CON. generates a string by joining together its two operand strings end-to-end. As an illustration,

```
'ABCD ' .CON. 'EFG' = 'ABCD EFG'
```

3.5.2 Substring

The substring operator generates a string by extracting a substring from its (string) operand. In the form

    [F1, F2]

this operator extracts the substring of length F2 beginning with character number F1 of the operand string. (The first character is character number 1.)

To illustrate the substring operator, consider the following:

```
'ABCDEF' [3, 2] = 'CD'

'XYZ' [3, 1] = 'Z'

'ABCD' [1, 4] = 'ABCD'
```

The two fields F1 and F2 of the substring operator may be any integer expressions. The F2 field may be omitted, in which case the substring from character F1 to the end is implied. For example,

```
'DOGCAT' [4] = 'CAT'
```

(Note: The symbols << and >> may also be used for [and], respectively.)

If the value of F2 is nonzero, then the substring defined by [F1, F2] must lie within the current bounds of its operand string; otherwise, execution is terminated with an "invalid substring" error. The following are <u>not</u> valid.

```
'ABC' [3, 2]

'ABC' [0, 2]

'ABC' [2, -1]

'ABC' [-1]
```

If the value of F2 is zero, then the substring operator is always valid and returns the null string. Additionally, S[F1] returns the null string whenever the value of F1 is greater than the length of S, and results in an "invalid substring" error if F1 is $\leq$ zero.

(Note: Assignment to a substring is discussed in Section 3.12.)

3.5.3 Relational Operators

The relational operators (=, <>, <, <=, >, >=) may be used with string operands. The result is either the integer 0 or the integer 1, just as for integer operands, as determined by the ASCII collating sequence (see Apendix III). Strings of unequal length are never equal.

Some examples illustrating these operators are given below.

```
'ABC' < 'ABD'

'ABC' < 'ABCD'

'ABC' = 'ABC'

'ABA' > 'AB1'

'123' < '124'

'+'   > '-'
```

3.6 STRING EXPRESSIONS

The string operators may have string expressions as operands. The precedence for unparenthesized expressions is (highest to lowest):

1. Substring

2. Concatenate

3. Relational operators

[Note that the result of a relational operation with string operands is of type integer, and hence a relational operation (even with string operands) is an integer expression.]

Thus, a string expression is:

1. A string constant, string variable (simple or subscripted), or string function call

2. A substring or concatenate operation, whose operands can be string expressions

3. A string expression enclosed in parentheses

Examples are given in later sections of Section 3.

3.7 ASSIGNMENT STATEMENT

The assignment statement for strings is similar to the assignment statement for integers. Its syntax is given by

```
<string variable> := <string expression>
```

No automatic conversion between string and integer exists in *SIMPL-Q*. Thus strings may not be assigned to integer variables and integers may not be assigned to string variables.

If the string represented by <string expression> has a length that does not exceed the maximum length of the variable, then the value of the variable is set to the value of <string expression>. If the value of <string expression> is too long for <string variable>, then the value of <string expression> is truncated to the maximum length of <string variable> before the assigment is made. For example, if S is declared by

```
STRING S[5]
```

and the assignment

```
S := '123456'
```

is made, then S will have the value '12345', regardless of the value of S before the assignment.

3.8 EXAMPLE

This example used the built-in function LENGTH, that returns the length of a string value. The following procedure, REMOVE, removes a given substring from a given string.

```
PROC REMOVE (STRING SUB, STRING STR)
 /* THIS PROC PRINTS THE RESULT OF REMOVING THE (SUB)STRING 'SUB'
    FROM THE STRING 'STR'. */
INT CHARPTR, FOUND
CHARPTR := 1
FOUND   := 0
WHILE CHARPTR + LENGTH(SUB) <= LENGTH(STR) + 1 .AND. .NOT. FOUND
  DO /* CHECK FOR OCCURRENCE OF 'SUB' BEGINNING AT 'CHARPTR' */
    IF STR[CHARPTR, LENGTH(SUB)] = SUB
      THEN /* FOUND */
        FOUND := 1
      ELSE
        CHARPTR := CHARPTR + 1
      END
  END /* LOOP */
IF FOUND
  THEN /* SUBSTRING 'SUB' IS AT POSITION 'CHARPTR' OF 'STR' */
  WRITE (STR[1, CHARPTR -1] .CON. STR[CHARPTR + LENGTH(SUB)])
  ELSE /* NO OCCURRENCE OF 'SUB' in 'STR' */
    WRITE(STR)
  END
/* END PROC 'REMOVE' */
```

3.9 I/O

READ and WRITE (and EOI) also may be used for strings. The rules for strings are similar to those for integers.

Strings to be read in must be indicated on the input medium just as a string constant would be indicated in a SIMPL-Q program (i.e., enclosed in apostrophes with any apostrophe in the string being indicated by two apostrophes). Commas and blanks or both may be used to separate input items, and strings and integers may be freely intermixed.

If the length of a string that is read in is greater than the maximum length of the string variable into which it is read, the input string is truncated to the maximum length of the variable. Thus, if S is a string variable,

```
READ(S)
```

with input

```
'ABCD'
```

would be completely equivalent to

```
S := 'ABCD'
```

Just as for assignments, no mixed types are permitted in a READ. Thus, for example, if X is an integer variable and the statement

```
READ(X)
```

is executed with

```
'345'
```

as the next input item, an error termination will occur.

WRITE will cause string expression values to be printed at predetermined tab positions, just as for integers. However, string values are left-justified in the columns, rather than right-justified as for integer values. If a string is too long for the current line, it will be continued on the following line.

3.10 EXAMPLE

```
/* THIS PROGRAM READS IN A LIST OF UP TO 99 NAMES AND PRINTS THEM
     OUT IN ALPHABETICAL ORDER.  THE NAMES MAY NOT BE MORE THAN 50
     CHARACTERS LONG. */

     STRING ARRAY IN[50](100) = ('') /* FOR HOLDING THE NAMES */
     INT N = 0,  /* NUMBER OF NAMES */
         I,      /* FOR GOING THROUGH 'IN' */
         SAVE,   /* FOR REMEMBERING A SPOT IN 'IN' */
         FINISHED = 0  /* SWITCH */

     ENTRY PROC SORT
     WHILE .NOT. EOI
       DO /* READ NAMES INTO 'IN' */
         N := N + 1
         IF N < 100
           THEN /* NOT TOO MANY */
             READ (IN(N))
         END
     END
   IF N > 99
     THEN /* TOO MANY NAMES INPUT */
       WRITE ('TOO MANY NAMES - ONLY FIRST 99 USED', SKIP)
       N := 99
     END
   WHILE .NOT. FINISHED
     DO /* PRINT OUT SORTED NAMES */
       SAVE := 0
       I :=1
       WHILE I <= N
         DO /* GO THROUGH AND GET FIRST NAME IN ALPHABETICAL ORDER */
           IF IN(I) <>''.AND. (SAVE = 0 .OR. IN(I) < IN(SAVE))
             THEN /* THIS NAME HAS NOT BEEN PRINTED, AND IT IS THE
               FIRST ONE FOUND OR SHOULD COME BEFORE THE CURRENT
               CANDIDATE */
             SAVE := I /* USE THIS ONE AS THE FIRST ALPHABETICALLY
                             SO FAR */
             END
           I := I + 1 /* GO ON TO NEXT NAME */
         END  /* LOOP TO GO THROUGH AND GET FIRST NAME IN ALPHA. ORDER */
       IF SAVE <> 0
         THEN /* A NAME WAS FOUND */
           WRITE (IN(SAVE), SKIP)  /* PRINT IT */
           IN(SAVE) := ''           /* MARK IT AS 'PRINTED' */
         ELSE /* ALL NAMES HAVE BEEN PRINTED */
           FINISHED := 1
         END
     END /* LOOP TO PRINT OUT NAMES */
   START
```

An example of the results of executing the program are indicated below.

```
Input:                              Output:

    'HERZOG'                             BUTLER
    'MCKISSICK'                          COOLEY
    'BUTLER'                             HERZOG
    'COOLEY'                             LIEBERSOHN
    'MANSFIELD'                          MANSFIELD
    'LIEBERSOHN'                         MCKISSICK
    'SCHMEISSNER'                        ROSE
    'ROSE'                               SCHMEISSNER
```

3.11 STRING FUNCTIONS

3.11.1 User-Defined Functions

A string function is a function whose value is a string (i.e., the function "returns" a string). The rules governing the use of string functions are analogous to the rules for integer functions (paragraph 2.1.2.2). The following illustrates the use of string functions.

Example

```
STRING FUNC ALPHABETIZE(STRING S)
/* THIS FUNC REARRANGES THE CHARACTERS OF STRING 'S' INTO ALPHABETICAL
   ORDER.  'S' MAY HAVE A MAXIMUM OF 256 CHARACTERS. */
STRING RESULT [256],  /* FUNCTION RESULT */
       NEXTCHAR[1]    /* NEXT CHARACTER IN ALPHABETICAL ORDER */
INT CHARNUM,  /* FOR LOOKING THROUGH CHARACTERS IN 'S' */
    NEXTNUM   /* POSITION OF 'NEXTCHAR' IN 'S' */
RESULT := ''
 WHILE S<>''
   DO /* EXTRACT NEXT CHAR (IN ALPHABETICAL ORDER) OF 'S' */
    NEXTCHAR := S[1,1] /* START WITH FIRST CHAR */
    NEXTNUM := 1
    CHARNUM := 2
    WHILE CHARNUM<=LENGTH(S)
      DO /* LOOK THROUGH CHARS OF 'S' FOR "SMALLEST" */
        IF S[CHARNUM,1] < NEXTCHAR
          THEN
            NEXTCHAR := S[CHARNUM,1]
            NEXTNUM := CHARNUM
          END
          CHARNUM := CHARNUM+1
        END /* LOOP TO LOOK THROUGH CHARS OF 'S' */
      RESULT := RESULT .CON. NEXTCHAR /* ADD NEXT CHAR TO RESULT */
      S := REMOVE(S,NEXTNUM)
    END /* LOOP TO EXTRACT NEXT CHAR */
  RETURN(RESULT)
```

```
/* END FUNC 'ALPHABETIZE' */

STRING FUNC REMOVE(STRING S, INT CHARNUM)
/*  FUNC TO REMOVE CHARACTER NUMBER 'CHARNUM' FROM 'S' */
RETURN (S[1,CHARUNUM-1] .CON. S[CHARNUM+1])
/* END FUNC 'REMOVE' */
```

3.11.2 Intrinsic Functions

In this section, some intrinsic (built-in) functions that facilitate programming with strings are described.

3.11.2.1 LENGTH

The function LENGTH returns the length of the value of its argument. The argument may be any string expression, and the result is of type integer.

Example

```
LENGTH ('ABC') = 3
LENGTH ('ABC' .CON. 'DE') = 5
```

3.11.2.2 MATCH

The MATCH function is used to find an occurrence of a substring in a string. The syntax is of the form

```
MATCH (S1, S2)
```

where S1 and S2 may be any string expressions. MATCH returns the character number in S1 of the first character of (the first occurrence of) the string S2. If S2 is not a substring of S1, then MATCH returns 0.

As an illustration, suppose that S = 'ABCATDOG' and T = 'AT'. Then

```
MATCH (S, T) = 4

MATCH (T, S) = 0

MATCH (S, 'A') = 1

MATCH (S, 'CAT') = 3

MATCH (S, 'DOGS') = 0

MATCH (S, 'CATS') = 0
```

3.11.2.3 INTF

INTF is used to convert a string of decimal digits (or a minus sign followed by decimal digits) into an integer value. If the string contains a character that is not a digit (other than a leading minus), then the program is terminated.

In the following example, let S1 = '123' and S2 = '017'. Then

```
INTF (S2) = 17

INTF (S1 .CON. S2) = 123017

INTF ('-' .CON. S2) = -17

INTF ('12345' [2,3]) = 234
```

3.11.2.4 STRINGF

STRINGF is the inverse of INTF. That is, STRINGF converts the value of an integer expression to string. As examples, let I = 22 and J = -15. Then

```
STRINGF (I) = '22'

STRINGF (J) = '-15'

STRINGF (I+J) = '7'

STRINGF (I-I) = '0'
```

The result of STRINGF is a string with no leading zeros. Thus, the length of the string returned by STRINGF is the number of significant digits in the integer value, plus 1 if the value is negative.

3.11.2.5 LETTERS

The function call LETTERS (<string expression>) returns a 1 (integer) if each character in the string is a letter (A-Z) and a 0 otherwise. The letters may be upper or lowercase (or both).

3.11.2.6 DIGITS

The value of DIGITS (<string expression>) is 1 if each character in the string is a digit (0-9) and 0 otherwise.

3.11.2.7 TRIM

The result of

```
TRIM (<string expression>)
```

is the value of <string expression> truncated to remove trailing blanks.

3.12 SUBSTRING ASSIGNMENT

The substring indicator may also be used on the left side of an assignment statement:

```
<string variable> [<first char>, <length>] := <string expression>
```

The rules for <first char> and <length> are the same as for the substring operator in string expressions (Section 3.5.2). The second field (, <length>) may also be omitted, just as for the string expression operator.

When the substring indicator is used in this manner, the substring specified by [<first char>, <length>] is replaced by the first <length> characters of the (string) value of <string expression>. The remaining characters of <string variable> are not changed. If needed, the value of <string expression> is extended on the right with blanks so that its length is not less then <length>.

To illustrate, let S1 = 'ABCDE' and S2 = '123456'. Then, after the assignments

```
S1 [2, 3] := 'XYZ'
S2 [2, 2] := '?'
S2 [5]    := '**XX'
```

the values of S1 and S2 will be

```
S1 = 'AXYZE'
S2 = '1? 4**'
```

Note that the length of the value of <string variable> cannot be changed by a substring assignment. If the current string length is zero, then no operation is performed. If <length> is not supplied and <first char> is greater than the current size (nonzero), an "invalid substring" error will be generated.

3.13 STRING PARAMETERS

Strings passed as arguments to (user) procedures and functions are passed by value unless otherwise specified as in Section 4.2. String arrays are passed by reference. These conventions are the same as for integers and integer arrays, as explained in paragraph 2.1.2.1.

The maximum length of a string parameter passed by value is set to the maximum length of the argument when the call occurs. Note that a string expression that is not a string variable (such as S.CON.T, S[I,J], or 'STRING CONSTANT') has a maximum length equal to the length of its value.

All arguments passed to intrinsic functions are by value. That is, an intrinsic function will not change the value of a parameter.

3.14 EXAMPLE

```
/* THIS PROGRAM REPLACES ALL SUBSTRINGS BETWEEN '/*' AND '*/' BY BLANKS */
STRING INPUT [80]
INT PTR1, PTR2
ENTRY PROC REMOVECOMMENTS
WHILE .NOT. EOI
  DO
    READ (INPUT)
    PTR1 := 1            /* INITIALIZE FOR SEARCH */
    WHILE PTR1 <> 0
      DO /* REMOVE SUBSTRINGS */
        PTR1 := MATCH (INPUT, '/*')
        IF PTR1 <> 0
          THEN /* FOUND BEGINNING */
            PTR2 := MATCH (INPUT, '*/')
            IF PTR2 > PTR1 + 1
              THEN /* FOUND END (AFTER BEGINNING) */
                INPUT [PTR1, PTR2 - PTR1 + 2] := '' /* BLANK IT OUT */
              END
          END
      END
    WRITE (INPUT,SKIP)
  END
/* END PROC 'REMOVECOMMENTS' */
START
```

For the input

```
'XXX /* COMMENT 1 */ YYY /* COMMENT 2 */'
'PTR1 := 1 /* INITIALIZE FOR SEARCH */'
'WHILE PTR1 <> 0'
'  DO /* REMOVE SUBSTRINGS */'
```

the program would produce the output

```
XXX YYY
PTR1 := 1
WHILE PTR1 <> 0
  DO
```

3.15 SPECIAL STRING TYPES

3.15.1 MODULE STRING

All SIMPL-Q programs must start with a MODULE STRING. The syntax has the form:

```
MODULE STRING name [length] = 'title'
```

The name (first six characters) of the string is taken as the name of the compile module. If there is more than one MODULE STRING, then only the first one is used and the others are ignored and an informative message is issued. If the MODULE STRING is initialized, then the string is used as page header for the compiler source listing.

3.15.2 ASCII STRING

The ASCII STRING type allows strings to be defined with nonstandard input characters. The syntax has the form:

```
ASCII STRING name [length] = 'characters or <int>'
```

where 'characters' is any character except < or >

"<" starts ASCII decimal constant mode (see Appendix III)

"int" is one or more decimal integers (not signed; 0-256) which may be separated by a blank or comma. Each integer will be treated as a binary character (18 bits).

">" terminates ASCII decimal integer mode.

Note: Characters cannot contain any "<" or ">" character.

Note that only integers base 10 are supported at this time. Therefore, <O'15'> is an error. Decimal integers greater than 256 do not generate an error and are converted to an 18-bit integer; however, the use of this fact is not recommended.

Example:

```
ASCII STRING BW [17] = 'ABC<10,11>ABC<10 11>AB<10>'
```

The compiler would produce the following code (in <u>octal</u>)

```
 15       Dope vector (13 current length) } String dope vector
 21       Dope vector (17 max length)     }
101       A
102       B
103       C
 12       <10>
 13       <11>
101       A
102       B
103       C
 12       <10>
 13       <11>
101       A
102       B
 12       <10>
  0
  .
  .
  .
  0
```

4. ADDITIONAL LANGUAGE FEATURES

4.1 ESCAPE MECHANISMS

4.1.1 EXIT Statement

The EXIT statement provides a means of escaping from a WHILE loop. In its basic form, the statement

```
EXIT
```

causes the immediate termination of the (innermost) WHILE statement containing the EXIT statement. Execution proceeds as if the WHILE statement has terminated normally.

The use of EXIT is illustrated by the following function.

```
INT FUNC FIND (INT NUMBER, INT ARRAY VALUES, INT SIZE)
/* FUNC TO RETURN THE SUBSCRIPT OF THE ELEMENT OF 'VALUES' HAVING
  VALUE 'NUMBER'.  If 'NUMBER' IS NOT IN 'VALUES', THEN 0 IS RETURNED.
  THE VALUES TO BE CHECKED ARE IN VALUES(1), ..., VALUES(SIZE). */
INT I
I := 1
WHILE I <= SIZE
  DO
    IF VALUES(I) = NUMBER
      THEN /* FOUND */
        EXIT
      ELSE
        I := I + 1
      END
  END
IF I > SIZE
  THEN /* NOT FOUND */
    I := 0
  END
RETURN(I)
/* END FUNC 'FIND' */
```

An exit of more than one level of nesting can also be performed by using an EXIT statement. To do so, the EXIT statement has the form

```
EXIT (<designator>)
```

where <designator> denotes the WHILE statement to be terminated. A <designator> is an identifier that is specified in the form

```
\<designator>\  WHILE ...
```

to designate the WHILE loop to be exited.

Consider the following program segment:

```
    \LOOP1\  WHILE I <= N  /* LOOP 1 */
        DO
         .
         .
         .
         WHILE 1  /* LOOP 2 (WILL NOT TERMINATE WITHOUT AN EXIT) */
           DO
             .
             .
             .
             IF I + J = K
               THEN
                 EXIT
               ELSE
                 IF I + J > K
                   THEN
                     EXIT (LOOP1)
                   END
               END
               .
               .
               .
           END /* LOOP 2*/
        I := I + 2
     END /* LOOP 1 */
   X := I
```

If the statement EXIT is executed, then the next statement to be executed would be I := I + 2 (which is in the WHILE statement designated by LOOP1). However, if the statement EXIT(LOOP1) is executed, the next statement to be executed would be X := I (the next statement after the WHILE loop designated by LOOP1).

A WHILE designator may be any identifier that has no other meaning in the segment (procedure or function) in which it is used.

4.1.2 RETURN Statement

The RETURN statement causes a return to the calling procedure or function. It may be any statement in a segment. The form

```
RETURN
```

is used for procedures, and the form

```
RETURN (<expression>)
```

is used for functions.

The function FIND of Section 4.1.1 may be rewritten to illustrate this statement:

```
INT FUNC FIND (INT NUMBER, INT ARRAY VALUES, INT SIZE)
INT I
I := 1
WHILE I <= SIZE
  DO
    IF VALUES (I) = NUMBER
      THEN /* FOUND */
        RETURN (I)
      ELSE
        I := I + 1
      END
  END
RETURN (0) /* NOT FOUND */
/* END FUNC 'FIND' */
```

Note that the last statement in a function need not be a RETURN (<expression>) if the structure of the function's statement list is such that a return is always made from within the statement list.

4.1.3 ABORT Statement

The statement

```
ABORT
```

causes the immediate (abnormal) termination of an entire SIMPL-Q program, regardless of the location of the ABORT statement or the depth of segment call nesting.

4.2 PARAMETER PASSING BY REFERENCE

Procedures may communicate scalar (integer or string) results through the parameters passed to it by specifying that a parameter is a reference parameter. Logically, this means that the scalar variable itself is passed to the procedure rather than the value of the variable, just as for array parameters. Thus, a procedure can then change the value of a variable in a CALL argument list.

A procedure declares a scalar parameter to be a reference parameter by means of the keyword REF. The following program illustrates the difference between normal parameter passing (by value) and reference parameters.

```
MODULE STRING TEST[4] = 'TEST'
INT X

PROC ADD1 (INT X, INT Y)
X := X + Y
```

```
PROC ADD2 (REF INT X, INT Y)
X := X + Y

ENTRY PROC MAIN
X := 3
CALL ADD1 (X, 2)
WRITE (X)
CALL ADD2 (X, 2)
WRITE (X)
START
```

This program would print

```
3     5
```

Note that only variables (simple or subscripted) may be passed by reference. That is, neither constants nor expressions (that do not consist of a variable only) may be passed by reference. (In particular, a substring may not be passed by reference.)

Functions may also have reference parameters.

4.3 RECURSIVE SEGMENTS

A segment that calls itself, either directly or indirectly, must be declared recursive. This is done by including the keyword REC before the segment definition. A segment that does not call itself can also be declared recursive in order to cause the dynamic (rather than static) allocation of locals (thus using no storage for the locals of the segment until the segment is invoked).

The following recursive function computes the factorial of an integer.

```
REC INT FUNC FACTORIAL (INT N)
IF N < 2
  THEN
    RETURN (1)    /* 0! = 1, 1! = 1 */
  ELSE
    RETURN (N* FACTORIAL (N-1))  /* N! = N* (N-1)! */
  END
/* END FUNC 'FACTORIAL' */
```

Currently in SIMPL-Q, all segments are compiled recursively. However, a segment that is intended to be used recursively should be declared REC.

4.4 ACCESS BETWEEN SEPARATELY COMPILED PROGRAM MODULES

It is often convenient to construct a program in two or more separately compiled modules rather than as a single compilation. The modules are compiled separately and then combined (see EASY II System Programmer's Guide) for execution. However, in this type of program construction it is often required that not only procedures and functions but also data in one module be accessible from within another module.

Since separate compilations are independent (that is, identifiers from one compilation are not known in any other compilation), special facilities are needed in order to provide the desired capabilities. In order for a SIMPL-Q (module 2) to access a procedure, function, or data from another module (module 1), two things are required:

1. The module (module 1) that contains the procedure, function, or data must make it available to other modules by specifying it as an entry point.

2. The module (module 2) that wishes to access the procedure, function, or data must specify that it is an external reference.

If these requirements are met, module 2 may use the identifier denoting the procedure, function, or data just as if the identifier were declared normally in module 2.

4.4.1 Entry Points

Segments and data in a SIMPL-Q program module may be made accessible to separately compiled program modules by declaring such a segment or data item as an entry point. This is done by preceding the usual declaration by the keyword ENTRY. The first PROC to be executed, since it is referenced from the operating system, must be declared an ENTRY PROC.

Examples

```
ENTRY INT I, J = 2

ENTRY STRING ARRAY S[10](20)

ENTRY REC PROC P(INT X) ...
```

Only global identifiers may be entry points. Due to an EASY restriction, entry point names may not exceed six characters in length. Truncation to six characters is performed if needed.

4.4.2 External References

In order to access an entry point of a separately compiled program module, the identifier to be accessed must be declared an external reference. The keyword EXT is used for this purpose.

External declarations for data items are similar to normal declarations. The differences are that initialization of externals is not allowed, and the size specification for strings and arrays may be omitted (since they are defined in another module). Examples of external data declarations are

```
EXT INT ARRAY VALUES

EXT STRING S, T

EXT STRING ARRAY SA[17](32)
```

External segment declarations must include a specification of the types of the parameters. Illustrations are:

```
EXT PROC P (INT, STRING)

EXT INT FUNC FIND (INT, INT ARRAY)

EXT PROC REMOVE (REF INT, STRING), INITIALIZE
```

External declarations may be global or local. Just as for entry points, an external name may not exceed six characters and truncation is used, if needed, to enforce this restriction.

4.4.3 Executable and Nonexecutable Modules

Any module which contains an ENTRY PROC is executable; all other modules are nonexecutable. A nonexecutable module must consist of (entry point) data items only; that is, it will not have any segments. A nonexecutable module must have at least one ENTRY declaration to be of value.

4.4.4 Example 1

The two modules given here illustrate the use of external references and entry points. The combined program reads in 50 numbers, sorts them into increasing order, and prints them.

Module 1:

```
MODULE STRING MODULE1[10] = 'MODULE1'
ENTRY INT ARRAY NUMBERS(50)
EXT PROC SORT
ENTRY PROC MAIN
READ (NUMBERS)
CALL SORT
WRITE (NUMBERS)
START
```

Module 2:

```
MODULE STRING MODULE2[10] = 'MODULE2'
EXT INT ARRAY NUMBERS
ENTRY PROC SORT
INT I, SAVE, LAST, INTERCHANGED
INTERCHANGED := 1
LAST := 49
WHILE INTERCHANGED
  DO /* BUBBLE SORT */
    I := 1
    INTERCHANGED := 0
    WHILE I <= LAST
      DO
        IF NUMBERS(I) < NUMBERS (I-1)
          THEN
```

```
                SAVE := NUMBERS (I-1)
                NUMBERS (I-1) := NUMBERS(I)
                NUMBERS(I) := SAVE
                INTERCHANGED := 1
              END
            I := I + 1
          END
        LAST := LAST -1
      END
    /* END PROC SORT */
    START
```

4.4.5 Example 2

The ability to access data in a separately compiled module is not absolutely necessary, since data can be accessed through argument lists. (However, the capability of data access between modules is needed for practical considerations.) As an illustration, consider the following modifications to the program in the previous example. The program obtained by combining modules 1A and 2A is equivalent to the program consisting of modules 1 and 2 above.

Module 1A:

```
    INT ARRAY INPUT(50)
    EXT PROC SORT(INT ARRAY)
    ENTRY PROC MAIN
    READ (INPUT)
    CALL SORT(INPUT)
    WRITE (INPUT)
    START
```

Module 2A:

```
    ENTRY PROC SORT(INT ARRAY NUMBERS)
    INT I, SAVE, LAST, INTERCHANGED
          .
          . same as module 2 above
          .
```

4.4.6 Executing a Program Having Multiple Modules

The procedure for executing one or more separately compiled modules varies slightly depending on whether the compiler on the CDC 6700 or the compiler on the QM-1 is being used. Appendix VIII contains a brief guide to the use of SIMPL-Q and examples. Additional information can be found in the EASY II System Programmer's Guide.

4.5 DEFINE FACILITY

A restricted macro capability exists in the compiler. There also exists a more capable macro facility (SIMPL MACRO LANGUAGE) as a precompilation pass ('M' option). Only the compiler facility will be discussed here; see Appendix VII for a description of the SIMPL MACRO LANGUAGE facility. This facility exists regardless of options specified. Macros are declared in a manner similar to that for other SIMPL declarations, and are invoked whenever the macro name is used as an identifier in the program. Macro parameters and nested macro calls (including recursive calls) are allowed.

A brief description of this facility follows.

4.5.1 Macro Definition

A macro definition has the syntax

```
DEFINE <define list>
```

where <define list> is a list of one or more definitions, separated by commas. Each definition has the form

```
<identifier> = <string constant>
```

where <identifier> is a normal SIMPL identifier, and <string constant> is a normal SIMPL string constant (enclosed in apostrophes). Macro parameters are denoted in the defining <string constant> by &n, where n is a digit between 1 and 9, inclusive, that refers to the argument number. The &n is replaced by the argument when the macro is invoked. Local define macros use stack space instead of allocated space.

4.5.2 Example

The following program (written for illustration only) prints the integers 1-10, *modulo* 4.

```
DEFINE
  INCR    = '&1 := &1+1',
  ASSIGN  = '&1 := &2',
  FOREVER = 'WHILE 1'
  MOD     = '&1-&1/&2*&2'
INT I=0, J
ENTRY PROC MAIN
FOREVER
  DO
    IF I >= 10
      THEN EXIT
      ELSE
        INCR(I)
        ASSIGN(J,MOD(I,4))
        WRITE(J)
      END
  END
START
```

4.5.3 Macro Expansion

Arguments to a macro are separated by commas and the argument list is enclosed in parentheses. Each argument may be:

1. A string constant (enclosed in apostrophes), in which case the value of the string constant is substituted for the formal parameter in the defining <string constant>.

2. Any string of characters not including a comma or right parenthesis, except that nested parentheses are allowed and a comma may appear between nested parentheses. In this case, the character string minus leading and trailing blanks is substituted for the formal parameters.

When a macro identifier is found during the processing of source text, the following occurs:

1. A copy is made of the macro definition.

2. The formal parameters in the copy of the defining string are replaced by the actual parameters from the argument list of the macro invocation.

3. The expanded macro then replaces the macro invocation (macro id and arguments list) in the source text, and processing of the source text resumes with the expanded macro.

4.5.4 Conventions and Restrictions

1. SIMPL COMMENTS (/* ... */) are removed from a <string constant> that defines a macro.

2. The usual scope rules apply for macro declaration. This means that macros may be defined as locals if desired. Note, however, that a global macro identifier cannot be redefined as a local without first turning off the macro expansion facility (see below) since the occurrence of the identifier in the local declaration list would invoke the global macro.

3. A macro invocation occurs whenever a macro identifier is found in the text. This means that a macro cannot be invoked within a string constant or comment, for example.

4. The argument list for a macro invocation is optional. Any empty list () is allowed.

5. An argument that is not a string constant may not cross an input line boundary. An argument list must begin on the same line as the macro identifier.

6. No more than nine formal parameters are allowed. Missing arguments are considered to be null, and arguments corresponding to no formal parameter are ignored.

7. The total length of all macro definitions (concatenated) cannot exceed 4000 characters.

8. An expanded marco plus the remainder of the line where the macro is invoked cannot exceed 400 characters. The length of all arguments concatenated cannot exceed 400 characters.

9. If an expanded macro plus the remainder of the line will not fit on one line, the expanded text is split as needed at the last blank before a line boundary.

10. No more than 50 macro invocations (including nested invocations) can occur from a single line of source text.

11. Note that neither ENTRY nor EXT may be used with DEFINE.

4.5.5 Options

The directives

```
/+ EXPANDOFF +/ and /+ EXPANDON +/
```

can be used to disable the expansion facility for any portion of source text. The expansion is initially ON. The directives

```
/+ EXPANDPRINTON +/ and /+ EXPANDPRINTOFF +/
```

can be used to obtain a listing of macro expansions. The print facility is initially OFF.

5. SOME SPECIAL-PURPOSE LANGUAGE FEATURES

5.1 CHARACTER DATA TYPE

5.1.1 Introduction

In order to facilitate a more efficient implementation of some string handling algorithms, SIMPL-Q has a third data type: character. A character is any ASCII character (see Appendix III).

In general, the addition of character data to SIMPL-Q involves mostly straightforward extensions of the integer and string handling concepts. Some of these extensions, and the variations that are needed for character data, are explained below.

5.1.2 Constants

Character constants are denoted by enclosing the character in quotation marks ("). A character constant may also be denoted by C '<integer constant>' where <integer constant> specifies the numerical value of the ASCII encoding of the character. The <integer constant> may have a maximum of 18 bits.

Examples of character constants are:

```
"A"     "1"     "?"     """"     C'13'     C'Ø'137''
```

5.1.3 Declarations

Scalar character declarations are similar to scalar integer declarations. Examples are:

```
CHAR C, C1 = "X"

ENTRY CHAR C2

EXT CHAR C3
```

Character array declarations are also similar to those for integer arrays. One difference is that a string constant may be included in the initialization list for a character array. The meaning is that the elements of the array are to be initialized with successive characters of the string. Examples are:

```
CHAR ARRAY CA1(10),

           CA(20) = ("A", 'CAT', "X")

ENTRY CHAR ARRAY B(10)

EXT CHAR ARRAY CA
```

In this example, CA2(1) is initialized to "C", CA2(2) to "A", and CA2(3) to "T".

5.1.4 Statements

The assignment statement has the form

```
<character variable> := <character expression>
```

for character data. Both sides of the assignment must be of type character.

The case statement can be used in the form

```
CASE <character expression> OF
\c₁\ <statement list>₁
          .
          .
          .
\cₙ\ <statement list>ₙ
{ELSE <statement list>ₙ₊₁} END
```

where $c_1,\ldots,c_n$ are character constants. The form of the character case statement is illustrated by

```
CASE NEXTCHAR OF
\"A"\ CALL CASEA
\"B"\ \"X"\ CALL CASEBX
\"?"\ CALL WHAT
 ELSE CALL OTHERCASE
 END
```

5.1.5 Operators

The only operators that may have character operands are the relational (=, <, etc.) operators. Both operands must be of type character, and the result is the same as it would be for single character strings consisting of the operand characters.

Thus, a character expression can only be a character variable (simple or subscripted), a character constant, or a character function call.

5.1.6 Intrinsic Functions and Procedures

5.1.6.1 INTVAL

```
INTVAL (<char expr>)
```

returns the integer whose value is the binary ASCII encoding of the character argument (see Appendix III). Examples are:

```
INTVAL ("A") = 65
```

INTVAL (" ") = 32

INTVAL ("a") = 97

5.1.6.2 CHARVAL

The result of

CHARVAL (<char expr>)

is of type character and is the inverse of INTVAL. Thus, for example,

CHARVAL (65) = "A"

CHARVAL (32) = " "

The value of the argument must have a value between 0 and 127, inclusive.

5.1.6.3 INTF

The INTF function also may be applied to character data. As examples,

INTF ("1") = 1

INTF ("9") = 9

The value of character argument must be one of the characters "0", "1",..., "9".

5.1.6.4 STRINGF

STRINGF also may have a character argument although no function is required for this conversion (see Section 5.1.8). The result is a string of length 1 consisting of the character. For example,

STRINGF ("A") = 'A'

STRINGF ("3") = '3'

5.1.6.5 CHARF

The function CHARF converts a string or integer argument to character. For string arguments, CHARF (<string expression>) is the first character of the string. For integer arguments,

CHARF (<int expr>) = CHARF [STRINGF(<int expr>)]

Thus, for example,

CHARF ('ABC') = "A"

CHARF (7) = "7"

CHARF (-17) = "-"

5.1.6.6 LETTER

LETTER is an integer function defined by

LETTER (<char expr>) = LETTERS [STRINGF(<char expr>)]

For example,

LETTER ("A") = 1

LETTER ("1") = 0

5.1.6.7 DIGIT

The integer function DIGIT is defined by

DIGIT (<char expr>) = DIGITS [STRINGF(<char expr>)]

For example,

DIGIT ("+") = 0

DIGIT ("1") = 1

5.1.6.8 UNPACK

UNPACK is an intrinsic procedure that stores the successive characters of a string into successive elements of a character array. The format is:

$\{$CALL$\}$ UNPACK (<string expr>, <char array>)

for example, if S = 'CAT' and A is a character array, then the statement

UNPACK (S,A)

would result in A(0) = "C", A(1) = "A", etc.

The string argument is extended on the right with blanks or truncated so that its length is the same as the number of elements in the array. Thus, every element of the array is given a value. Note that the string argument is first and the character array second.

5.1.6.9 PACK

The intrinsic procedure PACK is the reverse of UNPACK. The format is:

```
{CALL} PACK (<char array>, <string variable>)
```

All characters of the character array are used unless the maximum length of <string variable> is too small, in which case only the first k characters of the array are used, where k is the maximum length of <string variable>.

5.1.7 I/O

READ and WRITE are extended in a straightforward manner for character data, except as noted below. Characters to be read are enclosed in quotation marks just as for character constants in a SIMPL-Q program.

One difference in I/O for character data is that a string can be read into a character array. This works just as if the string were read into a string variable and then UNPACKed into the array. Similarly, a WRITE of a character array is the same as doing a PACK and a WRITE of the resulting string.

5.1.8 Characters as Strings

The set of characters is considered to be a subset of the set of strings. Thus, a character is also considered to be a string (of length 1) and may be used as a string without explicit conversion (STRINGF). Note that the converse is not true: no string may be used as a character without explicit conversion (CHARF).

5.1.9 Summary

Due to the special nature of character data, its usage has not been explained fully. It is expected that those who wish to use it will be advanced enough to extend the integer and string features logically to character. Other features not mentioned above, such as character functions, character parameters, etc., extend in a straightforward manner (e.g., characters are passed by value unless declared by reference, but character arrays are passed by reference).

5.2 BIT REPRESENTATION FOR INTEGER CONSTANTS

Integer costants may be specified in binary, octal, or hexadecimal, as well as decimal. However, these additional representations specify the bit pattern for the word in which the integer is stored, rather than the value of the integer. Thus, a maximum of 36 bits may be specified for the QM-1.

A bit representation consists of the letter B, O, or H, followed by the binary, octal, or hexadecimal, respectively, constant enclosed in quotes.

(Embedded blanks are permitted.) For example, integer value 23 can also be specified by any of the following:

```
B'10111'  B'010 111'

O'27'

H'17'
```

Similarly, -23 can be specified as O'777777777750' or H'FFFFFFE8'.

Trailing zeros may conveniently be specified by ending the constant in quotes by the letter Z followed by the decimal number of zeros to be included.

For example,

```
B'11Z3' = B'11000'

O'75Z6' = O'75000000'
```

A bit representation may occur anywhere in a SIMPL-Q program that an integer constant may occur. A bit representation may not be used for READ, however. No blanks may be embedded in a bit representation for an integer constant.

STRINGF may be used to convert an integer value to a string whose characters are the digits of a bit representation by using

```
STRINGF (<int expr>, <base indicator>)
```

where <base indicator> has value 2, 8, or 16. (10 is also permissible and is the default value.) Leading zeros are not included for base 10 in a result from STRINGF, but leading zeros are included for bases 2, 8, and 16 (string length is 18, 12, and 19). Similarly,

```
INTF (<string expr>, <base indicator>)
```

may be used to convert a string of binary, octal, or hex digits to integer.

5.3 BIT OPERATORS

5.3.1 Shift Operators

There are four shift operators in SIMPL-Q: left logical shift (.LL.), left circular shift (.LC.), right logical shift (.RL.), and right algebraic shift (.RA.). These are binary operators that are used in the form:

```
<integer expression> <shift operator> <shift count>
```

where <shift count> is an integer expression whose value is the number of bits to shift. Examples are:

```
O'3275Z4' .RA. 6 = O'3275Z2'
```

```
0'73Z10' .RL. 5 = 0'166Z8'

0'73Z10' .RA. 5 = 0'7766Z8' 50

0'77' .LL. 6 = 0'7700'

0'3275Z4' .LC. 15 = 0'275000000003'
```

5.3.2 Bit Logical Operators

The bit logical operators are complement (.C.), and (.A.), or (.V.), and exclusive or (.X.). Examples are:

```
.C. 0'1234567' = 0'777776543210'

B'110101' .A. B'11001' = B'010001'

B'110101' .V. B'011001' = B'111101'

B'110101' .X. B'011001' = B'101100'
```

5.3.3 Precedence

Bit complement (.C.) has the same precedence as the other unary operators but the binary bit operators have precedence over all other binary integer operators. Among the binary bit operators, the precedence (highest first) is:

```
.LL.  .LC.  .RL.  .RA.        shift

.A.       }
          }                   bit logical
.V.  .X.  }
```

5.4 PARTWORDS

The partword operator is similar to the substring operator. In the form:

```
<int expr> [F1,F2]
```

the partword operation has an integer value whose binary specification consists of the F2 bits begining at bit F1 of the value of the expression. (There is no sign extension of the leftmost bit.) Thus, the partword operator extracts F2 bits beginning at bit F1 (bit F1 is the leftmost bit of the partword extracted) and generates an integer value by adding leading zero bits to fill a word.

F1 and F2 may be any integer expression. Bit 0 is 2^0 (i.e., the rightmost bit). Thus, the values of F1 and F2 must satisfy all of the following:

$0 \leq F1 \leq 35$

$1 \leq F2 \leq 36$

The F2 field may be omitted, in which case F2 = F1 + 1 (the rest of the word).

As examples, consider the following:

17 [4,3] = 4

0'573201577123' [8,6] = 0'12'

0'573201577123' [17] = 0'577123'

Partwords may also be specified on the left of an assignment:

```
<integer variable> [F1,F2] := <integer expression>
```

In this case, F2 bits of the value of the variable, beginning with bit F1, are replaced by the rightmost F2 bits of the value of the expression. The remaining bits in the variable value remain unchanged.

For example, if integer variable X has value 0'6327', then after

```
X[8,6] := 0'7415'
```

X would have the value 0'6157'.

5.5 RECORD I/O

5.5.1 Introduction

Since I/O using READ and WRITE is not very flexible, there is also record-oriented I/O in SIMPL-Q. This allows a program to read input images into string variables and write one line of output text from a string expression. Thus, while record I/O is more primitive than stream I/O, it gives the user complete flexibility as to the format of input and output (although the user must scan the input images and build the output images).

5.5.2 READC

The intrinsic procedure READC may be used to read an entire input record (card, line, etc.) into a string variable. The syntax of a READC statement is:

```
{CALL}  READC({<skip>,} <readc item>{,<int variable>})
```

A <skip> for READC is the same as for READ, except that SKIP0 has no meaning for READC. The effect of a skip specification is different for READC, however, since a READC operation includes an implicit SKIP. Thus, for example, successive

```
READC (SKIP, S)
```

statements would read every other card.

A <readc item> may be a string variable, character array, or string array. These function as follows:

1. string variable - The next input record is read into the string variable. The input image is placed into the string just as it appears in the input (e.g., the character in the first card column becomes the first character in the string, etc.). All of the trailing blanks are removed. The input string is truncated, if needed, to the maximum length of the string variable.

2. character array - The characters of the input image are placed into successive elements of the array, beginning with element 0. If the input image is too long, it is truncated. If it is too short, it is padded with trailing blanks, so that the entire array is filled, unless the <int variable> is included.

3. string array - Successive input records are read into the (string) elements of the array. There must be enough input records to fill the array.

The optional <int variable> is designed for use with a character array. If included in a character array read, the array will not be padded out with blank characters. Instead, the integer variable will be set to the number of characters read.

If used with a string variable, the integer variable is set to the length of the image read. Note that this can also be obtained by using the LENGTH function after the READC. For a string array, the integer variable is set to the length of the last image read.

End of file for record input is determined by the intrinsic function EOIC, which is similar to EOI. The difference is that EOIC asks "Is there another input record?", while EOI asks "Is there another input item?"

READ and READC (and hence EOI and EOIC) are not designed to be intermixed, and a user does so only at his own risk.

5.5.3 WRITEL

The analogue to WRITE for record output is WRITEL. This intrinsic may be used to write out a string onto a line of output, with each string written on a new line. Thus, the statement

```
WRITEL(S)
```

is roughly the same as

```
WRITE(SKIP, S)
```

The syntax for a WRITEL is

```
        {CALL} WRITEL(<writel list>)
```

where <writel list> is one or more items, separated by commas. The items that may be used in <writel list> are:

1. a skip or eject specification - Note that SKIPO permits overprinting with WRITEL, and successive WRITEL (S, SKIP) statements would print on every other line.

2. a string expression - The string is printed on the next line, truncated to 132 characters if needed.

3. a character array - The array is PACKed and the resulting string is printed on the next output line.

4. a string array - The array elements (strings) are printed on successive output lines.

WRITEL and WRITE are not intended to be intermixed, although the problems are somewhat less severe than for intermixing READ and READC.

5.6 FILE I/O

5.6.1 Introduction

External data files can be used by a SIMPL-Q program. The files must be SIMPL-Q READF format if not generated by a SIMPL-Q program. Files must be pre-allocated using the QM-1 NOVA emulator (see EASY II System Programmer's Guide, Appendix D).

Logically, a SIMPL-Q file is considered to be a sequence (stream) of scalar data items. A previously created file can be used by a SIMPL-Q program. If no file with the proper name has been allocated, then the SIMPL-Q file routines will abort.

5.6.2 File Declaration

Files must be declared. A file declaration contains the keyword FILE followed by a list of identifiers. The identifiers are the (internal) file names to be used in the standard manner for EASY files. (File name identifiers may not exceed 10 characters in length.)

Files may also be declared as entry points or external references. A file may not be local unless it is an external reference. A file may be used as a parameter to a segment.

Examples of file declarations are:

```
FILE DATA1, DATA2

ENTRY FILE DATA3

EXT FILE DATA4
```

Normal buffer size for files is 512 (18 bit) QM-1 words. The buffer size can be changed but only in the compile module in which the file is defined (i.e., not defined in module as EXT FILE). Buffer size, if specified, is an integer constant indication of the number of QM-1 words in the buffer.

The syntax is: FILE <filename> = <buffersize>

<u>Example</u>

```
FILE TAPE1 = 1024
```

NOTE: Following buffer sizes are recommended for sequential, random, and word addressable files: default, 0, default.

Besides buffer size, the advanced user can specify additional information (QM-1 dependent) about the file. The syntax for this statement is:

```
FILE <filename> = {(<buffersize>, <xr>, <dcw>)}
```

where:

```
<buffersize> = integer constant; default 512
<xr> = exclusive access indicator
       1 read/write (default)
       0 multiple read (read only)
<dcw> = data control word (defined in the Nanodata TASK Reference
        Manual) to be used for disk access. Should be used
        with great care.
```

NOTE: <buffersize>, <xr>, and <dcw> are optional parameters.

The DCW "bits per word" value must be $\geq$ 4 and $\leq$ 18 (0 implies 18) and, if packed, $\leq$ 9. Furthermore, the bit count must divide evenly into 2304 (i.e., total # bits/sector). If the DCW does not conform to these conditions, a 0'1014' error will occur. When using alternate bit counts, care must be taken to ensure that the buffer is sufficiently large. The buffer will be flushed only when at least one sector worth of information is ready to be written to disk (except on Close or Rewind). The following table shows the legal DCW values and the minimum buffer size.

Legal DWC Width/P Values		Minimium Buffer Size 18-Bit Words
0/1**	unpacked	128.
4/1*	unpacked	576.
6/1*	unpacked	384.
8/1	unpacked	288.
9/1	unpacked	256.
12/1	unpacked	192.
16/1	unpacked	144.
18/1	unpacked	128.
4/0*	packed	64.
6/0*	packed	64.
8/0	packed	64.
9/0	packed	64.

Any other values for "width/p" will generate a 0'1014' error.

*If width $\leq$ 6, the hardware may generate lost data errors because the channel cannot access memory fast enough.

**A zero value for width implies a width of 18.

5.6.2.1 File Array Declaration

A file array declaration contains the keyword FILE ARRAY followed by a list of identifiers. The syntax is:

```
FILE ARRAY <filename> (<no. of elements>)
```

Example of a file array declaration:

```
INT I
INT ARRAY Z(64) = (0(64))
FILE ARRAY S(3)
/* THE ABOVE STATEMENT GENERATES
   3 FETS WITH
   INDEX   FILENAME
     0       'S0'
     1       'S1'
     2       'S2'          */
PROC INITIALIZE
/* THIS PROC WRITES 64 WORDS
   (36 BITS) OF ZEROS TO EACH OF THE THREE FILES
   S0, S1, S2              */
```

```
I:=0
WHILE I < 3 DO
WRITEF (S(I),Z)
I:=I+1
END
RETURN
```

As in a file declaration, buffer size may also be specified. The syntax is:

```
FILE ARRAY <filename> (<no. of elements>) = {<buffersize>{
```

Additional information may be specified about the file array by using the following syntax:

```
FILE ARRAY <filename> (<no. of elements> = {(<buffersize>,<xr>,<dcw>)}
```

5.6.3 READF

Files are read using READF. The syntax for this statement is:

```
{CALL{ READF (<filename>, <readf list>)
```

Each item in <readf list> may be:

1. An integer, string, or character variable.

2. An integer, string, or character array, in which case successive values are read into successive elements of the array.

The types of the items in the file must be compatible with the types of the items in the <readf list>. No error checking is performed.

End of file may be determined by EOIF, which has value 1 if all items have been read and value 0 otherwise. The syntax for this function is:

```
EOIF (<filename>)
```

5.6.4 WRITEF

Items are written into a file using WRITEF. The syntax

```
{CALL} WRITEF (<filename>, <writef list>)
```

is similar to that for READF. Each item in the <writef list> may be an expression of any data type or an array of any data type. WRITEF functions as a counterpart for READF.

5.6.5 Control Operations

A file is viewed logically as a sequence of items ending with a special end-of-file indicator. Two control operations are used in creating and positioning files.

ENDFILE is used to generate an end-of-file indicator. The statement

```
{CALL} ENDFILE (<filename>)
```

must be used after the last WRITEF statement that is used in generating the items in a file.

In order to return to the beginning of a file, REWIND is used. The statement

```
{CALL} REWIND (<filename>)
```

repositions the file at its first item.

5.6.6 Example

To illustrate the use of files, the following program reads in a set of integers, writes them into a file, and then reads them from the file and prints them.

```
INT NUMBER
FILE DATA
ENTRY PROC MAIN
WHILE .NOT. EOI
  DO /* READ IN NUMBERS AND CREATE FILE */
    READ (NUMBER)
    WRITEF (DATA,NUMBER)
  END
ENDFILE (DATA)
REWIND (DATA)
WHILE .NOT. EOIF(DATA)
  DO /* READ FROM FILE AND PRINT */
    READF (DATA,NUMBER)
    WRITE (NUMBER)
  END
START
```

5.6.7 Conventions and Restrictions

The sequence of file operations is important. The normal sequence for creating and then reading a file is illustrated in the example of Section 5.6.6. The restrictions are:

1. A READF may only follow another READF or a REWIND, unless it is the initial operation on an already existing file.

2. A WRITEF may be the first operation on a file or may follow a REWIND (or another WRITEF).

3. An ENDFILE may be the first operation or may follow a WRITEF or a REWIND.

4. A REWIND may only follow an ENDFILE, or another REWIND.

For these rules, EOIF is equivalent to READF.

The READF statements that read a file are completely independent from the WRITEF statements that create the file. A file is a sequence of scalar data items so that, for example, the elements of an array may be written out using WRITEF of an array and then read back in using I/O READF into scalar variables.

5.6.8 SIMPL-Q Random Access I/O

Random input/output procedures allow the user to create, access, and modify a multi-record file on a random basis without regard for their physical position or internal structure. Each record in the file may be read or written at random without logically affecting the remaining file contents. The length and content of each record is determined by the user.

Five system input/output procedures control the transfer of records between central memory and mass storage.

5.6.8.1 Random Access Open

$RAOPN opens the mass storage file and informs EASY that it is a random access file. The syntax is:

```
{CALL} $RAOPN (<filename>, <master index>)
```

If the file exists, the reopened master index is read from the file into the index array. The master index must be zeroed before initial use. Note, SIMPL-Q initially sets all global data items to zero.

<file name>	FILE
<master index>	INT ARRAY for master index

The following example prepares for random input/output on file SOURCE with a 10 (0-9) word master index. If the file already exists, the master index is read into the INDEX array.

```
FILE SOURCE
INT ARRAY INDEX (10)
         .
         .
         .
$RAOPN (SOURCE, INDEX)
```

5.6.8.2 Random Access Read

$RAR transmits data from the random access file to central memory. The syntax is:

```
{CALL} $RAR (<file name>, <data item>, <No. items>, <index>)
```

<data item>	<No. items>
integer	"1"
character	"1"
string	number of characters
integer array	number of array elements
character array	number of array elements
string array	number of array elements

<index> INT index into the master index array
$0 \leq$ index < length of the master index array

The following example reads from file SOURCE 20 items (indexed by 0th index) and places them in the INT ARRAY VALUES.

```
INT VALUES (20)
      .
      .
      .
$RAR (SOURCE, VALUES 20, 0)
```

5.6.8.3 Random Access Write

$RAW transmits data from central memory to the selected random access file. The syntax is:

```
{CALL} $RAW (<filename>, <data item>, <No. items>, <index>,
    <rewrite flag>)
```

where:

<rewrite flag>
- = 1 rewrite in place. Unconditional request; fatal error occurs if new record length exceeds old record length.
- = -1 rewrite in place if space available; otherwise, write at end of information.
- = 0 no rewrite; write normally at end of information.

The following example writes the 20 items in the INT ARRAY VALUES to FILE SOURCE (indexed by 2nd index) over-writing (if possible) the old information.

```
$RAW (SOURCE, VALUES 20, 2, 1)
```

5.6.8.4 Random Access Stindx

STINDX selects a different array to be used as the current index to the file. The syntax is:

```
{CALL} $RAS (<file name>, <index array>)
```

The call permits a file to be manipulated with more than one index. For example, when the user wishes to use a sub-index instead of the master index, STINDX is called to select the sub-index as the current index. The STINDX call does not cause the sub-index to be read or written; that task must be carried out by explicit $RAR or $RAW calls. It merely updates the internal description of the current index to the file. The STINDX index must be zeroed before initial use.

Examples:

```
INT ARRAY SUBIX (10)

{CALL} STINDX (SOURCE, SUBIX)
```

5.6.8.5 Random Access Close

$RACLS writes the master index from central memory to the file, and closes the file. The syntax is:

```
{CALL} $RACLS (<file name>)
```

The $RACLS call is optional for files that are only read, but must be used if file has been written to. Note: If the job aborts, an open random access file may not be closed.

Example:

```
$RACLS (SOURCE)
```

5.6.9 SIMPL-Q Word Addressable I/O

A word addressable file can be thought of as a random access memory which can be addressed to the word (i.e., as a normal computer memory).

5.6.9.1 Word Addressable Open

$WAOPN opens a file and informs EASY that it is to be handled as a word addressable file. The syntax is:

```
{CALL} $WAOPN (<file name>)
```

where

<file name> is an EASY file

Example:

```
     FILE CORE
           .
           .
           .
      $WAOPN (CORE)
```

5.6.9.2 Word Addressable Read

$WAGET reads the given <number> of words beginning at <word address> from the file into the item specified. The syntax is:

```
{CALL} $WAGET (<file name>, <item>, <number>, <word address>)
```

where

<file name>	FILE
<item>	INT or INT ARRAY in which data is placed.
<number>	Number of consecutive SIMPL-Q words to be read in. The <number> = 1 for an INT item. <number> cannot exceed the space in the array [i.e., 1 $\leq$ <number> $\leq$ length (array)].
<word address>	The starting word number in the word addressable file. The first word in a file is location 0 (i.e., 0 $\leq$ <word address> $\leq$ length of allocated file).

The file must contain information at the given location (word address); that is, the user must have written to location before reading. No checks are performed; junk will be returned if the user has not written to the file.

Example:

```
     FILE CORE
     INT ARRAY I(100)
             .
             .
             .
     $WAGET (CORE,I,100,1000)
```

Comment: Fill the array I with data from location 1000-1099 from file CORE.

5.6.9.3 Word Addressable Write

$WAPUT writes data from the <item> for the given <number> of words starting at the given <word address> in the file. The syntax is:

```
{CALL} $WAPUT (<file name>, <item>, <number>, <word address>)
```

where

<file name>, <item>, <number> and <word address> are the same as in $WAGET.

Example:

```
        FILE CORE
        INT ARRAY I(100)
                .
                .
                .
        $WAPUT(CORE,I,100,1000)
```

Comment: Write the array I to file CORE starting at word address 1000.

5.6.9.4 Word Address Close

$WACLS closes the word addressble file. The syntax is:

```
        {CALL} $WACLS (<file name>)
```

All word addressable files written to must be closed by the user. On a program abort, the file status may or may not reflect the last word addressable operation.

5.7 MULTIPLE INPUT-STREAM FILES

At present, EASY only supports single EOF files. SIMPL-Q aborts any program that checks for an EOF condition three times in a row.

5.8 OBTAINING THE EXECUTION TIME OPTIONS

The options specified on the processor-call statement that cause a SIMPL-Q program to be executed can be obtained by the program as it begins execution as described below.

The procedure initially invoked may have one parameter. The type of this parameter must be string array. If the procedure does have this parameter, then it will initially be passed a string array whose strings are the options specified on the control statement that invoked the program execution.

For example, if a program has the procedure

```
        ENTRY PROC MAIN (STRING ARRAY S)
        INT L
          .
          .
          .
        START
```

and is executed via the control statement

```
MAIN, SFM, INPUT, OUTPUT
```

then the value of the parameter S when the procedure MAIN is initially called will be

```
S(0) = 'SFM'

S(1) = 'INPUT'

S(2) = 'OUTPUT'

L:   = LENGTH(S)
```

Note: L would be equal to 3.

6. QM-1 DEPENDENT FEATURES

Most of the preceding SIMPL-Q language description is almost identical to the SIMPL-T (transportable version) language. Most programming can be performed using these features. But, the system programmer occasionally needs some special machine-dependent features. Being machine-dependent, however, means that these features may be subject to change in future versions of the language. In this area, the basic features were enhanced to include more arithmetic and logical support (extended precision), dynamic memory control, memory access, program control, string conversion, and Nanodata QM-1 micro operating system support. These enhancements were accomplished by adding a number of intrinsic procedures and functions to the language.

Many of the features discussed in this section are also discussed, along with additional background information, in EASY: The Design and Implementation of an Intermediate Language Machine.

6.1 ARITHMETIC AND LOGICAL SUPPORT

6.1.1 DIVIDE

This intrinsic procedure divides a 72-bit unsigned integer by a 36-bit unsigned integer and returns the dividend and the remainder.

```
{CALL} DIVIDE (<INT a1>, <INT a2>, <INT b>, <INT c>, <INT rem>)
```

<INT a> divided by <INT b> is returned in <INT c>; the remainder is returned in <INT rem>.

6.1.2 DSHIFTL

This intrinsic procedure does a logical shift on a 72-bit integer.

```
{CALL} DSHIFTL (<INT a>, <INT count>, <INT r>)
```

<INT a and a + 1> are shifted by <INT count> and returned in <INT r and r + 1> with zero fill.

A negative count indicates a right shift by the specified number of bits; a positive count indicates a left shift. If | count | >= 72, then all zeroes are returned.

Note: SIMPL-Q only supports a 36-bit integer; this intrinsic (also DADD and DSUB) assumes an integer of 72 bits (two SIMPL-Q integers). Care must be taken by the user to ensure that proper storage and referencing of double words (72 bits) occur.

Example:

```
INT A1, A2, C, R1, R2
A1 = 0
```

```
A2 = 1
DSHIFTL (A1, 36, R1) /* R1 = 1 and R2 = 0 */
```

6.1.3 DADD

This intrinsic procedure adds two 72-bit integers and returns the result.

```
{CALL} DADD (<INT a>, <INT b>, <INT r>)
```

<INT b and b + 1> are added to <INT a and a + 1> and the result is stored in <INT r and r + 1>.

6.1.4 DSUB

This intrinsic procedure subtracts a 72-bit integer from a 72-bit integer and returns the result.

```
{CALL} DSUB (<INT a>, <INT b>, <INT r>)
```

<INT b and b + 1> are subtracted from <INT a and a + 1> and the result is stored in <INT r and r + 1>.

6.1.5 MULTIPLY

This intrinsic procedure multiplies two unsigned 36-bit integers and returns a 72-bit result.

```
{CALL} MULTIPLY (<INT a>, <INT b>, <INT R1>, <INT R2>)
```

The low-order 36 bits are returned as integer R2, the high-order result is returned as R1.

6.2 DYNAMIC MEMORY CONTROL

6.2.1 PSLOAD

This intrinsic procedure adds program modules to the program space. The modules are specified by procedure name. The program space is managed internally by EASY (see PSFREE).

```
{CALL} PSLOAD (<PROC name>)
```

The program space is increased by the size of the load module. If program space overflows, a fault occurs. The module is read from the system library. If fatal I/O errors occur, a fault occurs. The module is checked to have valid entry and external linkages, and the proper header. Any error in format of the module results in a fault. Finally, the program space pointers are set to include the new module.

6.2.2 PSFREE

This intrinsic procedure deletes program modules from program space.

{CALL} PSFREE (<PROC name>)

The action is to delete the module containing <PROC name> and all modules that were loaded after it. The program space pointers are adjusted, and all external references to the deleted modules are reset to "unsatisfied".

No module with an active PROC can be deleted. Modules are deleted starting with the last loaded until the specified module is deleted or an active PROC is encountered. A PROC is 'active' if there exists at least one stack frame associated with the PROC.

6.2.3 $LINK

This intrinsic procedure allows the address (within a module) of an external (EXT) variable to be changed to another address.

{CALL} $LINK (<ext variable>, <new address>)

$LINK replaces the address portion of the EXT marker (see EASY design document for more discussion on EXT markers) with a new address, thereby linking that external variable to another address. Thus, future references to the external variable will reference the new location. $LINK (or $LINKNAME) should not be used unless necessary and fully understood by the user.

6.2.4 $LINKNAME

This intrinsic procedure allows the name of an external to be changed dynamically.

{CALL} $LINKNAME (<ext variable>, <new name>)

$LINKNAME changes the name in the EXT marker to the given name and zeroes the address portion of the EXT marker (makes the external unresolved). Therefore, on the next reference to the external variable, the system will resolve the reference based on the new name. The format of the new name must be in the EXT marker format (36-bit integer, up to six 6-bit ASCII characters left-justified, zero filled; if new name is a data item, as opposed to a PROC/FUNC, then the left bit should be on). Again, the user must fully understand this command before attempting to use it.

6.2.5 ALLOCATE

The ALLOCATE intrinsic enables the user, during run time, to allocate simple integers, characters, and strings as well as arrays of the previous types. Data structures can be generated as global variables (program space) or local variables (stack frame).

Possible errors that may occur when performing an ALLOCATE are as follows:

a. Invalid parameter values.

b. Allocated data structure may not fit in program or stack space.

c. Allocated data structure is larger than 2^{18}-1 QM-1 words.

d. Data structure name already exists in program space.

It is possible to perform a number of ALLOCATES using the same name for a local variable. The user should be aware that doing this will neutralize any operations done on or values put into the data structure between successive ALLOCATES. In other words, the user can access only the last variable of the same name allocated.

For review, the syntax of ALLOCATE is as follows:

```
ALLOCATE (<external name> {,<no. of elements>}
          {,<maximum string length>})
```

where

<external name> name of some external declared among the globals or locals of the module containing the ALLOCATE statement.

<no. of elements> required only when type of external is ARRAY in which case, it is equal to the dimension of the array.

<maximum string length> required only when type of external is STRING or STRING ARRAY.

Examples of global variable ALLOCATES:

```
MODULE STRING TEST [24] = 'ALLOCATE DATA STRUCTURES'
EXT INT INT1
EXT INT ARRAY IARY
EXT STRING STR1
EXT STRING ARRAY SARY
EXT CHAR CHR1
EXT CHAR ARRAY CARY
ENTRY PROC ALLOC
  INT M
  INT N
    N := 5
    M := 6
  CALL BUILD (N,M)
PROC BUILD (INT N, INT M)
ALLOCATE (INT 1)  /* ALLOCATE AN INTEGER */
ALLOCATE (IARY, N) /* ALLOCATE AN INTEGER ARRAY OF SIZE N */
ALLOCATE (STR1,M) /* ALLOCATE STRING WITH MAX. LENGTH OF M */
ALLOCATE (SARY, N, M) /* ALLOCATE STRING ARRAY WITH N ELEMENTS OF SIZE M */
ALLOCATE (CHR1) /* ALLOCATE A CHARACTER */
ALLOCATE (CARY, N) /* ALLOCATE A CHARACTER ARRAY OF SIZE N */
     .
     .
     .
```

Local variables would be done in a similar fashion except that the external declaration would be in the PROC performing the ALLOCATE.

6.2.6 MEMORY FILES

The seven intrinsics described under this heading provide a mechanism for the dynamic allocation and access of control store and mainstore data segments.

Each allocated file is preceded by a 12-word file header which is maintained and used by the intrinsics. The format of the file header is as follows:

Word No.	Bits	Definition
0	35-18	File name, characters 1 & 2
	17-0	File name, characters 3 & 4
1	35-18	File name, characters 5 & 6
	17-0	File name, characters 7 & 8
2	35-18	File name, characters 9 & 10
	17-12	Number of times file has been opened
	10	Availability (0=Available, 1=Unavailable)
	9	Openness (0=closed, 1=open)
	8	Nameness (0=nameless, 1=named)
	7-0	Residence (Disk(0), MS(1), CS(2))
3	35-0	Backward link to previous file header
4	35-0	Forward link to next file header
5	35-0	Size of file in QM-1 words (includes file header)

Restrictions do exist with regard to CS and MS files. These files are all of contiguous extent. Since EASY II is designed to allow direct I/O by a wide variety of target emulators and simulated I/O by access to memory extents allocated to target emulators, addresses within a file must be contiguous. Contiguous address ranges allow access rights checking (and mapping) to be performed by simple hardware (MS and CS) or firmware base and field length checking.

Although disk files are not yet supported, the syntax allows disk files to be specified as well as CS and MS files. In the future, it should be possible to reconfigure certain applications such that files may reside in CS, MS, or disk as desired, thus allowing performance/cost tradeoffs to be made. The syntax is as follows:

```
OPEN (<file> {,<file name>, <size>, <trouble>})
CLOSE (<file>)
GET (<file>, <item list>)
PUT (<file>, <item list>)
<position> := GETP (<file>)
PUTP (<file>, <position>)
GETL (<file>, <base>, <length>)
```

where

<file> = local file name declared by user with type "FILE" and stored in file name field at FET creation; e.g.,

```
FILE SPOOL
FILE ARRAY MAINSTORE(S)
FILE INPUT
```

<size> integer value specifying size of space needed

<trouble> reference integer receiving status of operation

0	everything's okay
1	no space for allocated file
2	hardware malfunction on directory or chain access
3	multi-read access error
4	write attempted on read only file
5	creation open with size = -1
6	file does not exist
7	specified directory not in working set
8	size = 0 for open on MS or CS for nameless file
9	file opened with size, existing file size
10	file opened with size, existing file size

<item list>	<integer> | <character> | <string> | <array> [,<item list>]
<position>	reference integer, used to set or receive the current position within the file
<base>	reference integer receiving base address of <file> not including file header
<length>	reference integer receiving length of structure referenced by <file>

The following describes the form and meaning of the EASY II file name string. The metasymbols are defined first, and the meaning of each are described next.

<file name> ::= '*<file string>*' -- *EASY II file name (a string)*

where

<file string> ::= <name> ::= $\{$A...Z $\cup$ 0...9 $\cup$ <NOVA Control System (NCS) spec>$^{+10}_{1}$
1 to 10 characters as allowed in the NCS file name where <NCS spec> is the set of special characters allowed by NCS in a file name (excludes a *)

::= <null> ::= no characters provided for name

::= <device> ::= CDR/CRT/KBD/LPT/MT<n>/<radevice>
devices under generalized (device-independent) I/O (some are not implemented)

::= <radevice> ::= CS/MS/DS<n><m>
control store, mainstore, disk, or disk directory drive and user numbers

::= <n><m> <n> = $\{0,1\}$, <m> = $\{0...9\}$

Files may be "connected to" or "created" by means of the OPEN. The following are the proper interpretations of "connect" and "create":

<null>	connect to a file which has for its permanent file name the local file name specified by the user. Residency specified by FET.
<name>	connect to a file, which has <name> as its permanent file name. Update FET name field from user local file name to <name>. Residency specified by FET.

<radevice>*	connect to file on random access device which has for its permanent file name the local file name specified by user
<radevice>*<name>	connect to file on random access device which has <name> as its permanent file name
*<device>	create a nameless (purged or closed) file on a specified device
<radevice>	create a file on a specified random access device which has for its permanent file name the local file name specified by user
<radevice><name>	create a file on a random access device with <name> as its permanent file name

The following paragraphs describe each of the intrinsics used to manipulate the memory files.

6.2.6.1 OPEN (<file>,<file name>,<size>,<trouble>)

Perform all necessary preliminaries to prepare FET File for I/O to file specified by <file name>. The FET residency and file name fields are updated based on <file name> semantics described previously. If the file does not exist and <file name> is properly specified, then the file is created. If an error occurs, a nonzero error code is returned in <trouble>.

<size> is interpreted as follows:

<size> = n if create, then allocate fixed length file of n-18 bit words

= 0 if create, then create variable extent file. If open existing file, then ignored

=-1 make file nameless, purged on close

6.2.6.1.1 OPEN (<file>)

Same as 6.2.6.1 with <size> = 0, <file name> = '' and abort with a message on an error.

6.2.6.2 CLOSE (<file>)

If the file is "nameless", purge from the disk directory or memory (CS, MS). Otherwise, just close the file, leaving the name in the directory or pool for possible future reference. All FET fields are properly set in anticipation for future opens.

6.2.6.3 GET (<file>, <item list>)

For each data type in the item list, read from the file the appropriate data (number of words based on size of data item) starting at current position in <file> and advancing to one past last word read. As many words as can be will be read until the end of file is reached. No dope vectors are assumed to be on the file with the exception of string dope vectors.

6.2.6.4 PUT (<file>, <item list>)

For each data type in the item list, write the appropriate data (excluding dope vector except for strings) to the file starting at the current position in <file> and continuing until the number of words specified by data type is transferred or end of file is reached. Position is set at one past last word written.

6.2.6.5 <position> := GETP(<file>)

This function returns the current position i in <file> where $0 \leq i \leq n-1$ in a file with n words.

6.2.6.6 PUTP(<file>,<position>)

Reset the position in file to i where $0 \leq i \leq n-1$. Values outside this range return as error code.

6.2.6.7 GETL (<file>,<base>,<length>)

This function returns the first word or base address (excluding file header) and length of <file>.

6.2.6.8 Memory File Intrinsic Examples

a. The following example opens a mainstore file, checks to see if the requested file was gotten successfully,and then determines the base and length of the file.

```
.
.
.
FILE ARRAY SPOOL(4)  /*  DECLARE SPOOL FILE ARRAY */
ENTRY PROC TESTA(STRING ARRAY PRAMS)
INT MSSIZE,BASE,LEN,TR
.
.
.
MSSIZE:= INTF(PRAMS(1),8) /* DETERMINE AMOUNT OF MAINSTORE*/
OPEN(SPOOL(SON), '*MS',MSSIZE,TR) /* TRY TO GET THE FILE*/
IF TR <> 0  /* WAS THE TROUBLE FLAG SET */
  THEN
    ABORTM ('CANT GET MAINSTORE, TROUBLE = '.CON. STRINGF (TR))
  ENDIF
GETL(SPOOL(SCON), BASE,LEN)  /* FIND OUT BASE AND LENGTH OF MAINSTORE
                                GOTTEN */
.
.
.
```

b. This example does OPEN'S, CLOSES'S, GET'S, PUT'S, etc.

```
MODULE STRING TSTFSP[30]='TEST FSP PROGRAMS'
EXT PROC $$IFSP
FILE A,B,C,D,E,F,U,V,W,X,Y,Z
FILE C1,G,H
FILE F1,F2,F3
INT POS(TF)
ENTRY PROC TEST
INT TA,TB,TC,TD,TE,TF,TU,TV,TW,TX,TY,TZ
INT TC1,TG,TH
OPEN(A,'*MS',15000,TA)
OPEN(U,'*CS',5000,TU)
OPEN(B,'*MS*BB',10000,TB)
OPEN(V,'*CS',1000,TV)
OPEN(C,'*MS*',5000,TC)
OPEN(D,'*MS',5000,TD)
OPEN(E,'*MS*EE',40000,TE)
OPEN(F,'*MS',2000,TF)
OPEN(G,'*MS*LAST',17900,TG)
OPEN(H,'*MS*LOOP',1900,TH)
OPEN(W,'*CS*',1000,TW)
OPEN(X,'*CS',1000,TX)
OPEN(Y,'*CS',1000,TY)      /*SHOULD NOT FIT IN MEMORY*/
 /* */
CALL TEST1
CALL TEST2
CLOSE(A)
 /* */
OPEN(C1,'MS*C',-1,TC1)
CLOSE(C1)
CLOSE(C)
OPEN(C1,'',-1,TC1)
CLOSE(C1)
CLOSE(H)
 /* */
CLOSE(D)
CLOSE(E)
CLOSE(F)
OPEN(E,'',-1,TE)
CLOSE(E)
CLOSE(G)
 /* */
CLOSE(Y)
CLOSE(X)
CLOSE(W)
OPEN(W,'',-1,TW)
CLOSE(W)
CLOSE(U)
CLOSE(V)
OPEN(U,'*CS',1500,TU)
OPEN(V,'*CS',1500,TV)
CLOSE(V)
CLOSE(U)
```

```
ENTRY PROC TEST1
STRING S1[20], S2[20], S3[20]
INT TF1,TF2,TF3
OPEN(F1, 'MS*BB',0,TF1)
S1 := 'THIS IS A TEST'
PUT(F1,S1)
POSITF1:= GETP(F1)
OPEN(F2,'CS*W',0,TF2)
S2 := ' FOR THE NEXT 30 '
PUT(F1,POSITF1)
PUT(F1,S2)
PUT(F2,S2)
OPEN(F3,'MS*EE',0,TF3)
S3 := ' SEC. YOU WILL HEAR '
PUT(F3,S3)
CLOSE(F1)
CLOSE(F2)
CLOSE(F3)
RETURN

ENTRY PROC TEST2
STRING P1[20],P2[20],P3[20],SNTNCE[60]
INT BASE,POS,LONG
OPEN(F1)
OPEN(F2)
OPEN(F3)
BASE:= 0
PUTP(F1,POSITF1)
GET(F1,POS)
GET(F1,SNTNCE)
PUTP(F1,BASE)
PUTP(F2,BASE)
PUTP(F3,BASE)
GET(F1,P1)
GETL(F1,BASE,LONG)
GET(F2,P2)
GET(F3,P3)
SNTNCE := P1 .CON. P2 .CON. P3
CLOSE(F1)
CLOSE(F2)
CLOSE(F3)
RETURN
START
```

6.3 SPECIAL MEMORY ACCESS

6.3.1 <u>LOC</u>

The absolute address of the item referenced via the parameter is returned as the value of this function. The parameter is any type.

Example:

```
LOC (<any item>)
```

6.3.2 $MEMORY

This intrinsic is an integer function which returns the contents of an EASY memory location.

```
$MEMORY (<n>)
```

The QM-1 word (18 bits, right-justified, not sign-extended) found at the QM-1 EASY address space location n, is returned.

6.4 SPECIAL PROGRAM CONTROL

6.4.1 RESTART

RESTART is used to escape from one procedure to a lower level parent. A RETURN returns contol to the caller, while RESTART returns control to the caller's caller or to a procedure even further down the stack. The PROC named as the parameter on RESTART specifies the target of the RESTART.

Example:

```
{CALL} RESTART (<PROC name>)
```

will return to PROC <PROC name> immediately after the last call executed in <PROC name>. This is equivalent to all the levels above <PROC name> executing RETURNs. If the PROC identified is not a parent of the PROC doing the RESTART, a fault occurs. If the RESTART target is the immediate parent, the result is equivalent to a RETURN (except no FUNC result returned).

6.4.2 DEADSTART

This intrinsic procedure performs a deadstart on the EASY machine as specified by a deadstart table. The deadstart table must be an INT ARRAY with a length of nine (9).

Example:

```
{CALL} DEADSTART (<array name>)
```

The following procedure contains several calls to DEADSTART.

```
ENTRY PROC $FAULD          /* DISK SEGMENT OF FAULT HANDLER */

CALL MESSAGE       /* DISPLAY DESCRIPTIVE MESSAGE */
IF MAINSTORE < REQMT THEN DEADSTART(DD$TBL) END
CALL OPTIONS       /* DISPLAY AVAILABLE OPTIONS */
CALL STEALKBD      /* GRAB THE KEYBOARD */

WHILE 1 DO
   CALL READKBD       /* READ A CHAR */
   CASE KBCHAR OF
     \"?" \ LONGOPTIONS
```

```
\"R"\  CALL $KBINIT
       DISPLAY((''))
       RESTART(MASTER)
\"Z"\  DEADSTART(DD$TBL)
\"0"\  CALL DUMPECBS
       DEADSTART(DD$TBL)
\"1"\  \"2"\  \"3"3\ CALL DUMPECBS
       CALL $TRACE(INTF(KBCHAR))
       DEADSTART(DD$TBL)
\"4"\  CALL DUMPECBS
       CALL $TRACE(3)
       CALL DUMPPS
       DEADSTART(DD$TBL)
\"Q"\  IF QOK<>''
         THEN CALL $KBINIT   /* RELEASE KEYBOARD */
           ECB(0) := POSTED
           SYSTEM(ECB$ARM,CTRLQECB,DBUGER)
           SYSTEM(ECB$PUT,CTRLQECB,ECB)
           RETURN      /* FIRE UP THE DEBUGGER */
         ELSE OPTIONS /*CANT DO A Q */
         END
  ELSE OPTIONS
  END
END
```

The deadstart table format is as follows:

Word bit:	35 — 18	17 — 0
0	Initial STR	Initial BS
1	TPS	Breakpoint Line
2	Breakpoint	Proc Name
3	3rd Library	Address
4	2nd Library	Address
5	1st Library	Address
6	Mainstore	Limit
7	Control store A limit	Control store B limit
8	CIO 1 | 2 | 3	directories 4 | 5 | 6

The Breakpoint Line Number (BLINE) should contain the image of the line instruction (i.e., line number shifted left logically six bits; or -1 to disable breakpoints). The mainstore and control store limits are the last addressable word. The 6-bit CIO directory settings contain an integer value indicating drive and user number (e.g., [illegible] 09 drive 0, 11-19 drive 1). If all six bits are ones, the entry is null.

6.5 SPECIAL STRING CONVERSION

6.5.1 $CONVST

This intrinsic function creates a string from consecutive QM-1 words. (Note: SIMPL-Q strings are composed of 18-bit characters - normally they will have 10 leading zeroes with an 8-bit ASCII character code in the right 8 bits).

```
$CONVST (<start>, <n>)
```

$CONVST returns a string of n QM-1 words (18 bits) starting at location start. Start is the absolute location (36 bits).

6.6 SYSTEM INTERFACE

6.6.1 SYSTEM

This intrinsic procedure allows the user to directly communicate with the Nanodata QM-1 micro operating system TASK.

```
{CALL} SYSTEM (<parameter 1>{,<parameters>})
```

The actions performed by this operator and the number and types of parameters are dependent on the value of the first parameter. The decoding of the SYSTEM operator is in this way unique. The SYSTEM operator can control EASY interrupt structure, basic I/O, subtasks, and hardware resources. A detailed description of the various instruction formats can be found in Section III.6.2 of reference 4.

6.6.2 SYSTIME

This intrinsic procedure allows the user to obtain a copy of the current system, time, and date.

```
{CALL} SYSTIME (<a>)
```

The reference integer <a> will contain the system time and date (in the format below) after the call to SYSTIME.

Format:

35 27	26 18	17 12	11 6	5 0
YEAR	DAY	HH	MM	SS

YEAR = binary year, i.e., 76

DAY = Julian day, $1 \leq DAY \leq (365 \text{ or } 366)$

HH = hour, $0 \leq HH \leq 23$

6.6.3 $EASY

This intrinsic procedure allows the user to create any EASY instruction in-line.

```
{CALL} $EASY (<opcode>{,<parameters>})
```

The opcode must be a constant and the other parameters (zero or more) are assumed to be operands and may either be variables or constants.

6.6.4 LIBRARY

This intrinsic procedure resets the PSLOAD library search order.

```
{CALL} LIBRARY (<lib1>{,<lib2>,<lib3>})
```

The library <lib1> will be searched first, <lib2> second, and <lib3> third by PSLOAD. When using this intrinsic, one should be aware that these libraries must be available in the directories which are then currently active.

6.7 INTERRUPT CONTROL

6.7.1 MIARM

This intrinsic procedure arms the interrupt system, causing interrupts to be allowed after MIARM has been executed.

6.7.2 MIDISARM

This intrinsic procedure disarms the interrupt system, causing interrupts to be locked out. Faults automatically rearm the interrupt system.

7. USING SIMPL-Q

7.1 SOURCE INPUT FORMAT

The normal scan of SIMPL-Q program text is the first 72 characters of each input record (e.g., card columns 1-72). This text is free-format and is essentially viewed as one continuous string of program text.

Keywords are reserved identifiers and may not be used as identifiers in a SIMPL-Q program. Keywords are listed in Appendix V.

Input record (e.g., card, teletype line) boundaries are meaningful only in that no keyword, identifier, integer or character constant, or symbol may be split across a record boundary. This restriction does not apply, however, to string constants and comments.

Comments can be nested; that is, a comment can contain other comments. Thus, a comment consists of all text between the characters /* and the first occurrence of the characters */ for which all occurrences of /* and */ in the text of the comment are themselves comment delimiters. (Thus, for example, it is always possible to temporarily remove a portion of a program by enclosing it in comment delimiters.)

Several compiler directives are available for program listing control, debugging aids, etc. Some of these may be specified by control card options (these are listed in Appendix I) and some may be specified by using a compiler directive in the source program text. A compiler directive is delimited by the characters /+ and +/, and may occur anywhere that a comment may occur (except within a comment).

The scan width for input text can be changed at any time by using the directive

```
/+ SCANLIMIT {<value>} +/
```

where <value> is the positive decimal integer number of the last character to be included as program text on each input record. The new scan limit becomes effective on the next input record after the one on which the directive occurs. If <value> is omitted, then 72 is assumed. (The initial value is also 72.) This feature is included primarily to allow the inclusion of information other than program text (e.g., sequence numbers) on the input records.

No validity checks are performed on <value>. Thus, for example,

```
/+ SCANLIMIT 1 +/
```

would render the remaining input text records useless.

7.2 DEBUGGING AIDS

7.2.1 Traces

7.2.1.1 Program Flow Traces (Not implemented at this time.)

Two traces for program flow are available: a trace of proc/func calls, and a line number trace. The call trace prints a message whenever a call to a procedure or function is executed and also prints a message when a return occurs. The messages include the names of the calling and called segments as well as the line numbers involved. (Only the first six characters of a segment name are printed.)

A line trace causes the number of a line to be printed when the statement on it is executed. The segment names (first six characters) are also printed as the segments are invoked.

The call and line traces can be activated by compiling with the T and Y options, respectively. They can also be activated and deactivated by using the compiler directives

```
/+ CALLTRACEON +/

/+ CALLTRACEOFF +/

/+ LINETRACEON +/

/+ LINETRACEOFF +/
```

These directives bracket the program statements for which a trace is to be activated. A call or line trace will be in effect for all statements between ON and OFF directives.

7.2.1.2 Variable Trace. (Not supported at this time.)

An execution trace for the value of variables is also available. This trace is activated by the directive

```
/+ TRACE <id list> +/
```

and is turned off by

```
/+ TRACEOFF <id list> +/
```

The <id list> is a list of identifiers, separated by blanks or commas; these identifiers must be known by the usual scope rules at the place where the TRACE or TRACEOFF occurs.

The TRACE and TRACEOFF directives bracket the part of the program for which the trace is to be performed. At execution, the name and value of a traced variable is printed after execution of an assignment statement in which the variable was the left side, and after execution of a call that passes the variable as an argument by reference. The line number of the statement is also printed.

An array trace will print values of elements used as scalars whose value may be changed and will include the value of the subscript. Arrays passed as arguments will be signaled by a message, but no values will be printed.

7. USING SIMPL-Q

7.1 SOURCE INPUT FORMAT

The normal scan of SIMPL-Q program text is the first 72 characters of each input record (e.g., card columns 1-72). This text is free-format and is essentially viewed as one continuous string of program text.

Keywords are reserved identifiers and may not be used as identifiers in a SIMPL-Q program. Keywords are listed in Appendix V.

Input record (e.g., card, teletype line) boundaries are meaningful only in that no keyword, identifier, integer or character constant, or symbol may be split across a record boundary. This restriction does not apply, however, to string constants and comments.

Comments can be nested; that is, a comment can contain other comments. Thus, a comment consists of all text between the characters /* and the first occurrence of the characters */ for which all occurrences of /* and */ in the text of the comment are themselves comment delimiters. (Thus, for example, it is always possible to temporarily remove a portion of a program by enclosing it in comment delimiters.)

Several compiler directives are available for program listing control, debugging aids, etc. Some of these may be specified by control card options (these are listed in Appendix I) and some may be specified by using a compiler directive in the source program text. A compiler directive is delimited by the characters /+ and +/, and may occur anywhere that a comment may occur (except within a comment).

The scan width for input text can be changed at any time by using the directive

```
/+ SCANLIMIT {<value>} +/
```

where <value> is the positive decimal integer number of the last character to be included as program text on each input record. The new scan limit becomes effective on the next input record after the one on which the directive occurs. If <value> is omitted, then 72 is assumed. (The initial value is also 72.) This feature is included primarily to allow the inclusion of information other than program text (e.g., sequence numbers) on the input records.

No validity checks are performed on <value>. Thus, for example,

```
/+ SCANLIMIT 1 +/
```

would render the remaining input text records useless.

7.2 DEBUGGING AIDS

7.2.1 Traces

7.2.1.1 Program Flow Traces (Not implemented at this time.)

Two traces for program flow are available: a trace of proc/func calls, and a line number trace. The call trace prints a message whenever a call to a procedure or function is executed and also prints a message when a return occurs. The messages include the names of the calling and called segments as well as the line numbers involved. (Only the first six characters of a segment name are printed.)

A line trace causes the number of a line to be printed when the statement on it is executed. The segment names (first six characters) are also printed as the segments are invoked.

The call and line traces can be activated by compiling with the T and Y options, respectively. They can also be activated and deactivated by using the compiler directives

```
/+ CALLTRACEON +/

/+ CALLTRACEOFF +/

/+ LINETRACEON +/

/+ LINETRACEOFF +/
```

These directives bracket the program statements for which a trace is to be activated. A call or line trace will be in effect for all statements between ON and OFF directives.

7.2.1.2 Variable Trace. (Not supported at this time.)

An execution trace for the value of variables is also available. This trace is activated by the directive

```
/+ TRACE <id list> +/
```

and is turned off by

```
/+ TRACEOFF <id list> +/
```

The <id list> is a list of identifiers, separated by blanks or commas; these identifiers must be known by the usual scope rules at the place where the TRACE or TRACEOFF occurs.

The TRACE and TRACEOFF directives bracket the part of the program for which the trace is to be performed. At execution, the name and value of a traced variable is printed after execution of an assignment statement in which the variable was the left side, and after execution of a call that passes the variable as an argument by reference. The line number of the statement is also printed.

An array trace will print values of elements used as scalars whose value may be changed and will include the value of the subscript. Arrays passed as arguments will be signaled by a message, but no values will be printed.

Note that only data identifiers may appear in <id list>. Thus, an array element may not be traced. If <id list> is empty in a TRACEOFF directive, all active traces are terminated.

7.2.2 Subscript Checking

Since the array subscript checking can be overlapped with computing the subscript, subscript checking is always performed.

7.2.3 Omitted Case Check (Not implemented at this time.)

Compiling with the D option causes checking for the occurrence of an unspecified case value in a CASE statement. If the expression value for the CASE statement does not correspond to any of the case numbers (or characters) and no ELSE part was specified, then a message is printed. This check can also be activated for portions of a program by using the directives

```
/+ CASECHECKON +/

/+ CASECHECKOFF +/
```

7.2.4 Conditional Text

In many instances, the best way to debug a program is to include extra statements in the program that provide information about the execution of a program as it executes. (An example of such a statement is a WRITE statement that prints the values of certain key variables.) Such statements are often somewhat cumbersome to put into a program at the right places, only to be removed after the bug has been found. The conditional text feature of the SIMPL-Q compiler provides a convenient means for handling such a situation.

The conditional text facility allows any string of source text to be either included or ignored as program text by the compiler. Such text is denoted by

```
/+ <indicators> <text> +/
```

where <indicators> is a string of digits and <text> is any SIMPL-Q program text that would be valid if the delimiters /+ and +/, and the <indicators> were removed. For example,

```
/+ 23 WRITE (X, Y, SKIP) +/
```

is an example of the conditional text

```
WRITE(X, Y, SKIP)
```

with indicators 2 and 3.

The "indicators" 0 - 9 are all initially off. To turn one or more indicators on, the directive

```
/+ SET <indicators> +/
```

is used. To turn indicators off,

```
/+ CLEAR <indicators> +/
```

is used.

Whenever conditional text is encountered by the compiler, the <text> is included in the program if, and only if, any of its <indicators> are on. Note that <text> need not be a complete statement. For example,

```
WRITE ('X=', X /+ 6, 'Y=', Y +/)
```

could be used to easily compile a program to print either the value of X only, or the values of both X and Y.

Note that a conditional compiler directive would be specified, for example, by

```
/+ 7 /+ LINETRACEON +/ +/
```

7.2.5 Abnormal Termination

Any premature termination of a SIMPL-Q program will cause the EASY fault handler to be invoked. A message is displayed on the CRT naming the line and procedure in which the fault occurred, the fault number, and a short description of the fault. The user is then offered several options for tracebacks and dumps (see EASY II System Programmer's Guide for details).

7.3 MESSAGES GENERATED BY SIMPL-Q

The messages generated by the compiler actually give the line number at which the error was discovered. (Thus, it is possible that the spacing of the text of a program can cause a line number to be given in a diagnostic message that is one or more lines after that on which the error occurred.) Similarly, splitting statements across card boundaries can sometimes make it difficult for the exact line number to be given in an execution-time diagnostic messsage.

The use of macros and DEFINEs can make it more difficult to determine the exact error. The standalone SIMPL Macro Language translator or /+ EXPAND PRINT ON+/ (for DEFINEs) may prove useful in determining the error.

Compiler error messages are normally self-explanatory. Error messages appear in three places in the output listing: after the macro pass (if "M" option is specified), after the source listing (if "S" option is specified), and after the EASY code listing (if "L" option is specified). Error messages normally do not terminate compilation (i.e., code will be produced). Therefore, the user is responsible for carefully examining the error message before executing the program. A few error messages are essentially warning messages. For example, SIMPL-T does not allow <procnames> to be passed as parameters. SIMPL-Q does allow this, but the compiler will generate a warning message.

One SIMPL-Q compiler message that may need clarification is "XXXX ADDRESS OUT OF RANGE.". This error message indicates that the "I" option (use indirect links for forward proc calls) was not used and a call for a proc which was more than 10000_8 locations (maximum direct address range) was found. The program should be recompiled with the "I" option.

Although SIMPL-Q treats the following three coding mistakes as minor errors, no error messages are generated: (1) using PROC for INT, (2) local names greater than six characters, and (3) missing CALL.

7.4 SOURCE LISTING

A source listing may be requested by using the S option. If the S option is not specified, no listing of the source program will be printed (even if the print directive is used) but diagnostic messages will be given. The N option may be used to suppress the printing of diagnostics.

Program listings include up to three numbers to the left of each line of source text. The first (leftmost) number is the line number. The second (if any) is the statement number for the first statement that begins on that line. The third number (if any) is the nesting level number for the first statement that begins on that line. The statement number and level number are omitted if no statement begins on that line.

Statements are numbered consecutively throughout a compilation, beginning with statement 1. The first statement in a procedure or function has nesting level 1, and the level increases by 1 inside a WHILE, IF, or CASE statement.

Several directives are available to control the printing of a source listing. The directives

```
/+ PRINTON +/

/+ PRINTOFF +/
```

may be used to print selected portions of a program. The directive

```
/+ EJECT +/
```

will cause the next line to be printed at the top of the next page. Similarly,

```
/+ SKIP  <count>  +/
```

or

```
/+ SPACE <count>  +/
```

will cause <count> blank lines to be skipped before printing the next line, where <count> is a positive decimal integer with 1 as the default.

If a listing control directive begins at the first character of a line of program text and if the line contains no text other than the directive, then that line will not be printed. Otherwise, the line will be printed as usual. For example, the line

```
/+ SPACE 2 +/
```

will not be printed, but the lines

```
/+ SPACE +/ X := 3

/+ SPACE 3 +/  Y := X + 2
```

will be printed.

7.5 ATTRIBUTE AND CROSS-REFERENCE LISTING

An attribute and cross-reference listing may be requested by using the F option or by including the directive

```
/+ ATTRIBUTES +/
```

in the program text. The attribute listing includes the characteristics, (relative) core address or internal number, and line number where defined for each identifier in the program. The cross-reference listing gives the line numbers where each identifier was used. If the value of a variable may be changed, an asterisk follows its line number. Note: Traceback dumps will give the local and global values for each active compile module.

7.6 KEYWORDS AND INTRINSIC IDENTIFIERS

Keywords (see Appendix V) may not be used as identifiers in a SIMPL-Q program. However, intrinsic identifiers (Appendix VI) may be redefined by the user. An intrinsic identifier is considered to be global. Thus, if a program redefines an intrinsic in a global declaration (including segment names), then that intrinsic cannot be used anywhere in the program. A local redefinition, however, only prohibits the use of the intrinsic in the segment containing the location redefinition.

7.7 OTHER OPTIONS

The B option can be used to turn off the debug aids that are otherwise performed. This would normally be used only on "debugged" "production" programs. For example, the B option turns off line numbers.

The R option causes no relocatable output to be generated. This is useful for doing syntax checks and generating listings when relocatable output is needed since it is faster than a full compile.

The I option informs the compiler to use an indirect link for all forward reference PROC/FUNC. At present, the forward direct link address range is 10000_8 words.

AD-A104 160 NAVAL SURFACE WEAPONS CENTER DAHLGREN VA F/G 9/2
SIMPL-Q REFERENCE MANUAL,(U)
MAY 81 C J NAPLES
UNCLASSIFIED NSWC-TR-81-262 NL

2 OF 2
AD
A104160

END
DATE
FILMED
10-81
DTIC

The directives

```
/+ RECURSIVEON +/

/+ RECURSIVEOFF +/
```

may be used to specify that all segments in a portion of a program are to be recursive, whether or not the keyword REC is included in the declarations. At present, all PROC/FUNC are recursive.

7.8 PROGRAM ANALYSIS FACILITIES

7.8.1 Program Statistics

If the directive

```
/+ STATISTICS +/
```

is included anywhere in the program text, the SIMPL-Q compiler will print statistical information about a program. The statistics include:

1. counts of the number of each type of statement (assignment, IF, etc.) used in the program

2. counts of the number of procedures, functions, and function calls

3. the average nesting level for statements in the program

4. the number of tokens generated for the program

5. the average number of tokens per statement

(A token is a syntactic entity, such as a keyword, operator, or identifier, that occurs in a program statement.)

7.8.2 Exeuction Statistics (Not implemented at this time.)

A statistical summary of program execution will be printed following the execution of a SIMPL-Q program if the directive

```
/+ EXECUTIONSTATISTICS +/
```

is included anywhere in the source text. The following are included in the statistical summary:

1. counts of the total number of times each type of statement (assignment, IF, etc.) was executed

2. counts of the number of times certain compound statement components (THEN parts, WHILE statement lists, etc.) were executed

3. counts of the number of times executed for the first statement in each statement list

4. maximum recursion level for each procedure or function that was actually called recursively. (The initial entry to a procedure is at level 0. The first recursive call is at level 1.)

Execution statistics for a program execution are printed, if requested, even if the execution is terminated by a program contingency.

It should be noted that the use of the execution statistics feature will significantly increase the size of most programs.

7.8.3 Execution Timing (Not implemented at this time.)

Execution timings for each procedure and function are provided if the directive

```
/+ TIMING +/
```

is included in the program text. Two timings are given:

1. CPU time excluding non-intrinsic calls. This represents the time actually spent in the code for a procedure or function, plus the time spent in library (intrinsic) calls.

2. CPU time including all calls. This represents the time from entry at recursion level 0 to exit at the *same* level.

The times given are in seconds, rounded to three places (msec.).

Since large fluctuations in timing can occur, depending mostly on system loading factors, several runs on the same data should be done, preferably when the system is not heavily loaded, in order to obtain more reliable results. These timings are intended for use in determining program bottlenecks and for most programs are accurate enough for that purpose. Procedures that execute for very short times (less than 1 msec.) are more likely to incur inaccuracies than are procedures that require more execution time.

The overhead required for execution timing is quite significant if the number of procedure and function calls is high. Since it is not unusual for execution times to be several times higher with timing, this feature should be used only when the timings are worth the extra cost in execution time.

7.8.4 Execution Statistics or Timing with Multiple Modules (Not implemented at this time.)

Execution statistics may be specified for any of the separately compiled modules of a program that use more than one separate compilation. Timing may also be specified for any module desired, except that if used in any of the modules, it must be specified for the module containing the START procedure designation in order to properly initialize the timing routines. Only those modules specified are monitored for statistics or timing.

7.9 MACRO PRE-COMPILE PASS (Simple Macro Language - SML)

The macro pre-processor described in Appendix VII has been incorporated into the compiler as on optional precompilation pass. The initial pass creates a source text file which is then fed to the compiler.

Some relevant information for the macro pre-processor is:

1. The macro pass is invoked by using the "M" option. If this option is not specified, no pre-compile pass will be performed. If the source contains "!<macro>" constructs, then the 'M' option must be used.

2. If the macro pass is done, the input file on the SIMPL-Q card denotes the SML source file. There is no way to create a SIMPL-Q source file or element by using the macro pre-compilation pass.

3. If the macro pass is done, all line numbers (source listing and diagnostics) will refer to the macro source, rather than the generated SIMPL-Q source.

4. The "S" option generates a listing of the SIMPL source only. To list the macro source, the macro directive !OPTION(LIST) must be used.

8. ADDITIONAL NOTES ON THE IMPLEMENTATION OF SIMPL-Q

8.1 SIMPL-Q OBJECT CODE

The SIMPL-Q compiler produces code for an intermediate language machine called EASY. (See the EASY Design Manual for details on the EASY machine.) EASY executes as a virtual machine on the Nanodata QM-1 micro-programmable computer. The "L" option will list the EASY code produced by the compiler.

The SIMPL-Q compiler can either run on the CDC 6000 SCOPE 3.4 system (cross compiler) or on the Nanodata QM-1 EASY system. Appendix VIII contains a brief discussion on how to use the SIMPL-Q compiler on either system. Appendix I contains a brief discussion on how to execute a SIMPL-Q program on the Nanodata QM-1 system. (See the EASY System Programmer's Guide for more details.)

8.2 INTERFACE WITH OTHER LANGUAGES

Currently, SIMPL-Q can directly communciate with the Nanodata TASK O/S and other virtual machines via system functions.

8.3 SOME COMMENTS ON EFFICIENCY

This section contains some random comments regarding the efficiency of certain SIMPL-Q features.

1. Recursive procedures and functions incur relatively little additional overhead. It may well be reasonable, in fact, to declare nonrecursive segments that use a large amount of local storage as recursive in order to avoid the static allocation of the local storage. A possible exception here is that local string arrays in a recursive segment require the initialization of the dope vectors for the elements at entry, and this could prove costly if a recursive segment is invoked often. At present, all PROC/FUNCs are compiled as recursive.

2. The passing of a string argument by value (the default) means that the string must be copied into the called segment, whereas an argument passed by reference is not copied. This is unlikely to be significant unless segments with value string parameters are very heavily used.

3. If the value of a logical operation can be determined from the first operand only, then the second will not be evaluated. For example, for the operation

$$\langle expr\rangle_1 \ .OR. \ \langle expr\rangle_2$$

if $\langle expr\rangle_1$ is nonzero, then $\langle expr\rangle_2$ will not be evaluated. Thus, the operands in a sequence of logical operations should be in the order that would usually determine the result most quickly, if possible. At present, all expressions are evaluated.

4. Hardware partword operations are used (if possible) when a partword operator specifies a (constant) half-word. For example:

```
A[35, 18] or A[17]
```

8.4 FUNCTION WITH SIDE EFFECTS

Functions are assumed to have no side effects. Thus, some function calls may be eliminated in order to optimize the code generated. For example, function calls involved in an unevaluated operand of a logical operation (see Section 8.3) and successive function calls whose arguments are unchanged need not be made under the assumption of no side effects.

Those who write functions that have side effects should ensure that the elimination of function calls by an optimization process will not adversely affect their program.

REFERENCES

1. Nanodata Corporation, *QM-NCS, Preliminary Systems Operations Guide*, Williamsville, New York, May 1977.

2. Nanodata Corporation, *Task Control (TCP 1.05)*, Revision 2, Williamsville, New York, 1976.

3. John G. Perry, Jr., *EASY II System Programmer's Guide*, Naval Surface Weapons Center, Dahlgren Laboratory Technical Report NSWC/DL TR-8198, Dahlgren, Virginia (in press).

4. Charles W. Flink, II, *EASY - The Design and Implementation of an Intermediate Language Machine*, Naval Surface Weapons Center, Dahlgren Laboratory Technical Report NSWC/DL TR-3765, Dahlgren, Virginia, December 1977.

5. Control Data Corporation, *CDC INTERCOM Reference Manual*, CDC 603071000, St. Paul, Minnesota, 1974.

6. John G. Perry, Jr., *An Informal Investigation into Software Engineering as Applied to a Compiler Bootstrapping Project*, Naval Surface Weapons Center, Dahlgren Laboratory Technical Report NSWC/DL TR-3625, Dahlgren, Virginia, April 1977.

7. Charles W. Flink, II, *EASY - An Operating System for the QM-1, MICRO-10 Proceedings*, October 5, 1977.

8. Victor R. Basili, *SIMPL-X: A Language for Writing Structured Programs*, U. of Maryland Computer Science Center, TR-223, January 1973.

9. IBM Corp., *IBM System/360 Operating System, PL/I (F), Language Reference Manual*, GC 28-8201-4, 1972.

10. Donald E. Knuth, *The Art of Computer Programming: Fundamental Algorithms*, Volume I, Addison-Wesley, 1969.

11. J. A. Vernon and R. E. Noonan, *A High-Level Macroprocessor*, U. of Maryland Computer Science Center, TR-297, March 1974.

12. V. R. Basili and A. J. Turner, *SIMPL-T: A Structured Programming Language*, U. of Maryland Computer Science Center, Computer Note CN-14.2, August 1975.

APPENDIX I

EXECUTING A SIMPL-Q PROGRAM ON THE QM-1

EXECUTING A SIMPL-Q PROGRAM ON THE QM-1

The following information is for illustrative purposes. More detailed information is contained in the SIMPL-Q User's Guide, Appendix VIII.

```
SIMPLQ, <compiler options>, <source file>, <binary file>, <schema file>
.
.        SIMPL-Q program
.
```

The compiler options are:

A - Go even if severe errors are found.

B - Turn all debug aids off (line numbers).

D - Check for omitted case.

F - Generate attribute and cross-reference listing.

I - Use indirect link for forward PROC/FUNC calls.

L - Print object code.

M - Pre-compilation macro pass.

N - Suppress printing of diagnostics.

R - Do not generate a relocatable element.

S - Print source listing.

X - Abort if any diagnostic occurs.

The default compiler options are SFM. The default source file is *CDR, the card reader. The default binary file is the file QM1. The default schema file is SCHEMA. SIMPL-Q compiler is on Library SIMPLQLIB.

Note that the sequence of control cards given above applies only to the situation in which only one execution is to be done in a run. If additional compilations and executions are performed, then the BIND (or EDITLIB) control statement must be included before TEST or MASTER can be used.

APPENDIX II

PRECEDENCE OF OPERATORS

PRECEDENCE OF OPERATORS

The SIMPL-Q operators are listed below in order of precedence from highest to lowest.

[]	part
.C. .NOT. - (unary)	unary
.RA. .RL. .LL. .LC.	shift
.A.	bit logical
.V. .X.	
* /	arithmetic
+ - (binary)	
= <> < > <= >=	relational
.AND.	logical
.OR.	
.CON.	string

APPENDIX III

ASCII CHARACTER CODES

ASCII CHARACTER CODES

Octal Code	Character	Card Code	Octal Code	Character	Card Code
000	NUL		040	SP	blank
001	SOH		041	!	12-7-8
002	STX		042	"	7-8
003	ETX		043	#	3-8
004	EOT		044	$	11-3-8
005	ENQ		045	%	
006	ACK		046	&	12
007	BEL		047	'	5-8
010	BS		050	(	12-5-8
011	HT		051	)	11-5-8
012	LF		052	*	11-4-8
013	VT		053	+	12-6-8
014	FF		054	,	0-3-8
015	CR		055	-	11
016	SO		056	.	12-3-8
017	SI		057	/	0-1
020	DLE		060	0	0
021	DC1		061	1	1
022	DC2		062	2	2
023	DC3		063	3	3
024	DC4		064	4	4
025	NAK		065	5	5
026	SYN		066	6	6
027	ETB		067	7	7
030	CAN		070	8	8
031	EM		071	9	9
032	SUB		072	:	2-8
033	ESC		073	;	11-6-8
034	FS		074	<	12-4-8
035	GS		075	=	6-8
036	RS		076	>	0-6-8
037	US		077	?	0-7-8

ASCII CHARACTER CODES

Octal Code	Character	Card Code	Octal Code	Character	Card Code
100	@	4-8	140		
101	A	12-1	141	a	
102	B	12-2	142	b	
103	C	12-3	143	c	
104	D	12-4	144	d	
105	E	12-5	145	e	
106	F	12-6	146	f	
107	G	12-7	147	g	
110	H	12-8	150	h	
111	I	12-9	151	i	
112	J	11-1	152	j	
113	K	11-2	153	k	
114	L	11-3	154	l	
115	M	11-4	155	m	
116	N	11-5	156	n	
117	O	11-6	157	o	
120	P	11-7	160	p	
121	Q	11-8	161	q	
122	R	11-9	162	r	
123	S	0-2	163	s	
124	T	0-3	164	t	
125	U	0-4	165	u	
126	V	0-5	166	v	
127	W	0-6	167	w	
130	X	0-7	170	x	
131	Y	0-8	171	y	
132	Z	0-9	172	z	
133	[	12-2-8	173	{	
134	\	0-2-8	174	|	
135	]	11-2-8	175	}	
136	^		176	~	
137	_		177	DEL	

NOTE: 1. On IBM 029 EBCDIC keypunches four characters are different:

QM-1 Character	IBM Character
[	¢
]	!
!	|
\	0-2-8 (upper case 'T')

2. Four \ (0-2-8) in Card Columns 1-4 is an EOF.

APPENDIX IV

FORMAL SPECIFICATION OF SIMPL-Q SYNTAX

FORMAL SPECIFICATION OF SIMPL-Q SYNTAX
(Basic SIMPL-T Language Only)

1. Program

```
<program> ::= {<declaration list>} {<segment list>} start {<identifier>}
```

2. Declarations

```
<declaration list> ::= {<declaration list>} <declaration>
<declaration>      ::= <variable declaration> |
                       <structure declaration> |
                       <external declaration>

<variable declaration> ::= <integer declaration> |
                           <string declaration> |
                           <char declaration>

<integer declaration> ::= {entry} int  <int dec list>
<int dec list> ::= {<int dec list>,} <int dec item>
<int dec item> ::= <identifier> {=<signed constant>}

<string declaration> ::= {entry} string <string dec list>
<string dec list> ::= {<string dec list>,} <string dec item>
<string dec item> ::= <identifier> [<constant>] {=<string constant>}

<char declaration> ::= {entry} char <char dec list>
<char dec list> ::= {<char dec list>,} <char dec item>
<char dec item> ::= <identifier> {=<char constant>}

<structure declaration> ::= <array declaration>
<array declaration> ::= <int array declaration> |
                        <string array declaration> |
                        <char array declaration>

<int array declaration> ::= {entry} int array <int array dec list>
<int array dec list> ::= {<int array dec list>,} <int array dec item>
<int array dec item> ::= <identifier> (<constant>) {=(<int array init list>}
<int array init list> ::= {<int array init list>,} <int array init item>
<int array init item> ::= <signed constant> {(<constant>)}
<string array declaration> ::= {entry} string  array <string array dec list>)
<string array dec list> ::= {<string array dec list>, } <string array dec item>
<string array dec item> ::= <string spec> (<constant>) { =(<string array init list>)}
<string spec> ::= <identifier> [<constant>]
<string array init list> ::= {<string array init list>,} <string array init item>
<string array init item> ::= <string constant> {(<constant>)}

<char array declaration> ::= {entry} char array <char array dec item>
<char array dec list> ::= {<char array dec list>, } <char array dec item>
<char array dec item> ::= <identifier> (<constant>) {=(<char array init list>)}
<char array init list> ::= {<char array init list>,} <char array init item>
<char array init item> ::= <char constant> {(<constant>)} | <string array init item>

<external declaration> ::= ext {other} proc <external segment list> |
                           ext {other} <type> func <external segment list> |
                           ext int <identifier list> |
                           ext string <string spec list> |
                           ext char <identifier list> |
                           ext int array <array list> |
                           ext string array <string array list> |
                           ext char array <array list>
<external segment list> ::= {<external segment list>,} <identifier> {(<type list)}

<identifier list> ::= {<identifier list>,} <identifier>
<string spec list> ::= {<string spec list>, } <identifier> {[<constant>]}
<array list> ::= {<array list>, } <identifier> {(<constant>)}
<string array list> ::=
          {<string array list>,} <identifier> {[<constant>]} {(<constant>)}

<type> := int   string   char
<type list> ::= {<type list>,} <type list item>
<type list item> ::= {<type call>} <type>   <type>  <type struct>
<type struct> ::= array
<type call> ::= ref
```

3. Program Segments

```
<segment list> ::= {<segment list>} <segment definition>
<segment definition> ::= <proc definition> | <func definition>

<proc definition> ::= <proc heading>  <segment body> {return}
<func definition> ::= <func heading> {<segment body>} {return (<expr>)}
<proc heading> ::= {other} {entry} {<rec>} proc <identifier> {(<parameter list>)}
<segment body> ::= {<local declaration list>} <statement list>
<func heading> ::=
          {other} {entry} {<rec>} <type> func <identifier> {(<parameter list>)}
<parameter list> ::= {<parameter list>,} <parameter>
<parameter> ::= {<type call>} <type> <identifier> | <type> array <identifier>

<local declaration list> ::= {<local declaration list>,} <local declaration>
<local declaration> ::= <local variable declaration>|
                        <local structure declaration> |
                        <external declaration>
<local variable declaration> ::= <type> <identifier list>

<local structure declaration> ::= int array <array bound list>|
                                  string array <string array bound list>|
                                  char array <array bound list>
<array bound list> ::= {<array bound list>,} <identifier> (<constant>)
<string array bound list> ::=
          {<string array bound list>,} <string spec> (<constant>)
```

4. Statements

```
<statement list> ::= {<statement list>} <statement>
<statement> ::= <assign stmt>| <if stmt> | {\<exit designator>\} <while stmt>|
          <case stmt> | <call stmt> | <exit stmt> | <return stmt>

<assign stmt> ::= <extended int variable> ::= <expr> |
                  <extended string variable> ::= <string expr> |
                  <char variable> ::= <char expr>

<if stmt> ::= if <expr> then <then clause> {else <else clause>} end
<then clause> ::= <statement list>
<else clause> ::= <statement list>
<while stmt> ::= while <expr> do <while clause> end
<while clause> ::= <statement list>
```

```
<case stmt> ::= <integer case stmt> |<char case stmt>
<integer case stmt> ::= case <expr> of <case list> {else <else clause>} end
<case list> ::=  {<case list>,} <case form>
<case form> ::= <case designator> <statement list>
<case designator> ::= {<case designator>} \<integer>\
<char case stmt> ::=
          case <char expr> of <char case list> {else <else clause>} end
<char case list> ::= {<char case list>,} <char case form>
<char case form> ::= {<char case designator>} <statement list>
<char case designator> ::= {<char case designator>} \<char constant>\

<call stmt> ::= call <identifier> {(<actual parameter list>)}
<actual parameter list> ::= {<actual parameter list>,} <actual parameter>
<actual parameter> ::= <expr> |<string expr>|<char expr> |<array identifier>

<exit stmt> ::= exit {(<exit designator>)}
<return stmt> ::= return {(<expr>)}
<exit designator> := <identifier>
```

5. Expressions

```
<expr> ::= {<expr> .OR.} <logical product>
<logical product> ::= {<logical product> .AND.} <relation>
<relation> ::= {<relation> <relational op>} <simple expr> |
               <string expr> <relational op> <string expr> |
               <char expr> <relational op> <char expr>
<simple expr> ::= {<simple expr> <add op>} <term>
<term> ::= {<term> <mult op>} <bit sum>
<bit sum> ::= {<bit sum> <bit or op>} <bit product>
<bit product> ::= {<bit product> .A.} <shift>
<shift> ::= {<shift> <shift op>} <factor>
<factor> ::= <unary op> <factor> |<part primary>
<part primary> ::= <primary> {<part designator>}
<primary> ::= <constant> |<variable> |<function designator> |(<expr>)
```

```
<relational op> ::= = |<> |> |< |>= |<=
<add op> ::= + |-
<mult op> ::= * |/
<bit or op> ::= .V. |.X.
<shift op> ::= .RA. |.RL. |.LL. |.LC.
<unary op> ::= .NOT. |.C. | -
<part designator> ::= [<first bit> {,<bit count>}]
<first bit> ::= <expr>
<bit count> ::= <expr>

<function designator> ::= <identifier> {(<actual parameter list>)}
<variable> ::= <identifier> {(<expr>)}
<constant> ::= <integer> |<binary> |<octal> |<hexadecimal>
<signed constant> ::= {-} <constant>
<array identifier> ::= <identifier>
<integer> ::= {<integer>} <digit>
<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<binary> ::= B'<binary form> {<trailing zeros>}'
<binary form> ::= {<binary form>} <binary character>
<binary character> ::= 0 | 1
<octal> ::= O'<octal form> {<trailing zeros>}'
<octal form> ::= {<octal form>} <octal character>
<octal character> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7
<hexadecimal> ::= H'<hexadecimal form> {<trailing zeros>}'
<hexadecimal form> ::= {<hexadecimal form>} <hex character>
<hex character> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
          A | B | C | D | E | F
<trailing zeros> ::= Z<integer>

<identifier> ::= {<identifier>} <letter> | <identifier> <digit>
<letter> ::= A | B | C | D | E | F | G | H | I | J | K | L | M | N | O|
          P | Q | R | S | T | U | V | W | X | Y | Z | $
<string expr> ::= {<string expr> .CON.} <string part primary>
<string part primary> ::= <string primary> {<substring designator>}
<string primary> ::= <string>|<string function designator>|(<string expr>)|
          <char expr>
```

```
<substring designator> ::= [<first char> {,<char count>}]
<string> ::= <string variable> |<string constant>
<string variable> ::= <identifier> {(<expr>)}
<string constant> ::= '<string form>' |''
<string form> ::= {<string form>} <character>
<first char> ::= <expr>
<char count> ::= <expr>
<string function designator> ::= <function designator>

<char expr> ::= <char constant> |<char variable> |<char function designator>
<char constant> ::= "<character>"
<char variable> ::= <identifier> {(<expr>)}
<character> ::= any single legal character
<char function designator> ::= <function designator>
<extended int variable> ::= <variable> {<part designator>}
<extended string variable> ::= <variable> {<substring designator>}
```

APPENDIX V

KEYWORDS

KEYWORDS

The following are reserved keywords and may not be used as identifiers in a SIMPL-Q program.

ABORT
ASCII
ARRAY
CALL
CASE
CHAR
DADD
DEFINE
DIVIDE
DO
DSHIFTL
DSUB
ELSE
END
ENTRY
EXIT
EXT
FILE
FUNC
IF
INT
LOC
MODULE
MULTIPLY
OF
OTHER
PROC
PSFREE
PSLOAD
REC
REF
RESTART
RETURN
START
STRING
SYSTEM
SYSTIME
THEN
WHILE
$CONVST
$EASY
$LINK
$LINKNAME
$RACLS
$RAR
$RAOPN
$RAS
$RAW
$WACLS
$WAGET
$WAOPN
$WAPUT

APPENDIX VI

INTRINSIC PROCEDURES AND FUNCTIONS

INSTRINSIC PROCEDURES AND FUNCTIONS

An intrinsic (built-in) procedure or function identifier as well as an I/O control parameter identifier (EJECT, SKIP, etc.) may be redefined by the user if desired. Note that if an intrinsic identifier is redefined by the user, then its intrinsic meaning is lost. However, an intrinsic that is redefined by a local declaration is lost only to the segment containing such a local definition.

VI.1 INTRINSIC PROCEDURES

(An intrinsic procedure call need not include the keyword CALL.)

Name	Section	Arguments	Function
ABORT	4.1.3	none	terminate program abnormally
ALLOCATE	6.2.5	<ext name>, <int>, <int>	dynamically allocates integers, characters, strings, or arrays
CLOSE	6.2.6.2	<file>	close a memory file
DADD*	6.1.3	<int>, <int>, <int>	72-bit integer add
DEADSTART	6.4.2	<int array>	deadstarts the EASY machine
DIVIDE*	6.1.1	<int>, <int>, <int> <int>, <int>	72-bit integer divide
DSHIFTL*	6.1.2	<int>, <int>, <int>	72-bit integer shift
DSUB*	6.1.4	<int>, <int>, <int>	72-bit integer subtract
ENDFILE	5.6.5	<file>	generate end-of-file
GET	6.2.6.3	<file>, <item list>	read from memory file
GETL	6.2.6.7	<file>, <int>, <int>	returns base address and length of memory file
LIBRARY	6.6.4	<lib>, <lib>, <lib>	resets the library search sequence
MIARM	6.7.1	none	enables interrupts

*QM-1 dependent

Name	Section	Arguments	Function
MIDISARM	6.7.2	none	disables interrupts
MULTIPLY*	6.1.5	<int>, <int>, <int>, <int>	72-bit integer multiply
OPEN	6.2.6.1 6.2.6.1.1	<file>, <string>, <int>, <int>	opens a memory file
PACK	5.1.6	<char array>, <string>	pack char array into string
PSLOAD*	6.2.1	<proc>	load proc
PSFREE*	6.2.2	<proc>	free proc
PUT	6.2.6.4	<file>, <item list>	write to memory file
PUTP	6.2.6.6	<file>, <int>	reset position in memory file
READ	2.7.1 3.9 5.1.7	see indicated sections	stream input
READC	5.5.2	see Section 5.5.2	record input
READF	5.6.3	<file>, <item list>	file input
REWIND	5.6.5	<file>	reposition file
RESTART*	6.4.1	<proc>	restart program at proc
SYSTEM*	6.6.1	<int>, <item list>	Nanodata TASK system call
SYSTIME	6.6.2	<int>	fetch time and date
UNPACK	5.1.6	<string expr>, <char array>	unpack chars of string into array
WRITE	2.7.2 3.9 5.1.7	see indicated sections	stream output
WRITEF	5.6.4	<file>, <item list>	file output
WRITEL	5.5.3	see Section 5.5.3	record output

*QM-1 dependent

Name	Section	Arguments	Function
$EASY*	6.6.3	<int>, <item list>	generates EASY instruction
$LINK*	6.2.3	<item>, <int>	resolve external address
$LINKNAME	6.2.4	<item>, <int>	change external name
$RACLS*	5.6.8.5	<file>	random close
$RAR*	5.6.8.2	<file>, <item>, <int>, <int array >	random read
$RAOPN*	5.6.8.1	<file>, <int array>	random open
$RAS*	5.6.8.4	<file>, <int array>	random store index
$RAW*	5.6.8.3	<file>, <data item>, <int>, <int>	random write
$WACLS*	5.6.9.4	<file>	word addressable close
$WAGET*	5.6.9.2	<file>, <item>, <int>, <int>, <int>	word addressable read
$WAOPN*	5.6.9.1	<file>	word addressable open
$WAPUT*	5.6.9.3	<file>, <item>, <int>, <int>	word addressable write

VI.2 INTEGER FUNCTIONS

Name	Section	Arguments	Result
DIGIT	5.1.6	<char expr>	indicates whether char is a digit
DIGITS	3.11.2	<string expr>	indicates whether all chars of string are digits
EOI	2.7.1	none	indicates whether all items have been read

*QM-1 dependent

Name	Section	Arguments	Results
EOIC	5.5.2	none	indicates whether all records have been read
EOIF	5.6.3	<file>	indicates whether all file items have been read
GETP	6.2.6.5	<file>	returns memory file position
INTF	3.11.2 5.1.6 5.2	see indicated sections	argument converted to integer
INTVAL	5.1.6	<char expr>	ASCII binary value of char
LENGTH	3.11.2	<string expr>	(current) length of string
LETTER	5.1.6	<char expr>	indicates whether char is a letter
LETTERS	3.11.2	<string expr>	indicates whether all chars are letters
LOC*	6.3.1	<item>	return address
MATCH	3.11.2	<string expr>$_1$, <string expr>$_2$	position of string$_2$ in string$_1$
$MEMORY*	6.3.2	<int>	return value of a memory location

VI.3 STRING FUNCTIONS

Name	Section	Arguments	Result
STRINGF	3.11.2 5.1.6 5.2	see indicated sections	argument converted to string
TRIM	3.11.2	<string expr>	string with trailing blanks removed
$CONVST*	6.5.1	<item>, <int>	create string from memory

QM-1 dependent

VI.4 CHARACTER FUNCTIONS

Name	Section	Arguments	Result
CHARF	5.1.6	see Section 5.1.6	argument converted to character
CHARVAL	5.1.6	<integer expression>	inverse of INTVAL

*QM-1 dependent

APPENDIX VII

SIMPL MACRO LANGUAGE

SIMPL MACRO LANGUAGE

VII.1 INTRODUCTION

Appendix VII was written to provide a user's manual and technical description of a macro processor developed at the University of Maryland. It is currently implemented on the Nanodata QM-1 in a NSWC/DL systems language called SIMPL-Q (Perry, 1977), and provides the user capabilities for string substitution in any symbolic text. The syntactic design of the macro facilities is based on SIMPL, and the reader is assumed to be familiar with the SIMPL language. The facilities provided are modeled after the PL/I compile time facilities (IBM Corp., 1970).

VII.2 SIMPL MACROS

VII.2.A Introduction

"A macro is a specification of a pattern of instructions and/or pseudo-operators which may be frequently repeated within a program" (Knuth, 1968, p. 626). This implementation scans as input text for macro invocations and produces an updated output text. Input text consists of any string of characters including macro invocations and definitions. Each macro defined by the user or the system will have some text as defined above associated with it; the definition itself is replaced by the null string in the output text.

As macro invocations are encountered within the input text, the invocation is removed and the corresponding text for the invoked macro is substituted. After the text of the macro is completely scanned, examination of the input will continue from the character immediately following the call (blanks included). Note that in rescanning the macro text, other macro invocations, definitions, or anything which is legal in the normal input may be encountered. This could result in several levels of recursive invocations.

The macros themselves are similar to those found in most modern assembly languages. They consist of text and parameters (Note: Throughout this appendix the word 'text' will retain the meaning presented in the preceding paragraph.) Parameters may be passed to user-defined macros and are handled in a special way as described in Section VII.2.B. Parameters are again any legal text string and are assigned specific values at macro invocation time.

All system-defined macros have replacement texts of the null string. Of these, only the one called !NULL may be redefined by the user, and all others are to be considered as reserved names.

All macro and parameter invocations are preceded by the special character !. The following notation will be adopted for further discussion.

<TEXT>	implies any string (possibly empty) of characters including macro calls and definitions

<EXPR>	is any arithmetic expression consisting solely of compile time variables, constants, special parameter variable (see Section VII.7), and operators as defined in Section VII.E. A complete BNF* of expressions can be found in Section VII.7
<NAME>	is any 1 to 12 character alphanumeric name with the first character being alphabetic
<PARAMETER>	is a text

When encountered in text (macros, parameters, etc.), spaces are treated as any other non-alphanumeric character and they will be reproduced in the output text. Name collection ends only upon encountering the first non-alphanumeric character so two names must be separated by at least one such character, possibly a blank. Within compile time expressions, blanks are treated only as delimiters.

VII.2.B Macro Definition and Usage without Parameters

User macros may be defined as follows:

```
!MACRO <NAME> = <TEXT> !END
```

This causes the macro assiociated with NAME to be assigned the corresponding text <TEXT>. In collecting the text of the macro, other macro calls may be encountered. These macros are immediately processed and their processed text, not the macro call, will become part of the new macro (see Example 3). Advanced features, however, permit macro calls within the body of a macro; i.e., a call that will be invoked only when the macro is invoked (see Section VII.7).

After macros are defined, they may be later invoked (in the simplest case) via the syntax:

```
!<NAME>
```

or

```
!<NAME>( )
```

where <NAME> is the name of the macro. (A discussion of macro invocation with parameters occurs in Section VII.2.C.) The text associated with the macro is substituted in the output at the point where the invocation is encountered.

```
EX 1:  !MACRO ADDR=INSTRUCTION [18,18]!END
```

This defines a macro 'ADDR' with the replacement text:

```
INSTRUCTION[18,18]
```

*Backus-Naur Form

if the input:

```
        !ADDR := J
```

was encountered, then the corresponding output would be:

```
        INSTRUCTION[18,18] := J
```

The text for any macro may continue over card boundaries and card separations are remembered. Consider the following example:

```
        EX 2:   !MACRO PUSH=
             PTR := PTR + 1
             IF PTR > SIZE THEN CALL OVERFLOW END
             !END
```

and its usage:

```
        A := A + 1 !PUSH STK (PTR) := A
```

would produce the output lines:

```
        A := A + 1                               )
        PTR := PTR + 1                            } from macro expansion
        IF PTR > SIZE THEN CALL OVERFLOW END     )
        STK (PTR) := A
```

When macro invocations are encountered within a definition, they are immediately expanded as explained earlier.

```
        EX 3:  !MACRO A = S(5)  !END
               !MACRO B = X := !A  !END
```

causes the body of macro B to be:

```
        X := S(5)
```

and not the string:

```
        X := !A
```

It is possible to define a macro as in the latter case above, but such an explanation is suspended until Section VII.3.A on macro nesting.

Collection of the macro name terminates with the first non-alphanumeric character. This causes some difficulties if you wish some macro expansion text to immediately precede an alphanumeric character.

```
        EX 4:  !MACRO A = IDENTIFIER !END
               !A1 := !A1 + 1
```

Normally this would invoke the macro called A1, but the intended purpose is to call A and then follow it by the string 1. This may be done by calling A with the following alternate syntax:

```
!A( )1 := !A( ) 1 + 1
```

This would then output the text string:

```
IDENTIFIER1 := IDENTIFIER1 + 1
```

VII.2.C Macros with Parameters

Up to nine parameters may be passed to macros at invocation time. Parameters, like macros, may consist of any legal text including other macros. Also, as in the case of macro definitions, if other macros are encountered during parameter collection, they are immediately expanded and the expanded text, not the macro call, is used to define the parameter body.

Parameters are passed via the following syntax:

```
!<NAME> (<PARAMETER1>,<PARAMETER2>,...,<PARAMETER9>)
```

where <NAME> is the name of some macro and <PARAMETER I> is the text of parameter number I. Only the parameters invoked in the macro body need be present in the macro call.

```
EX 5:   !BUMP(VAR)
```

The above invocation calls a macro named BUMP and passes as parameter 1 the string VAR.

```
EX 6:   !MACRO A=XYZ!END
        !BUMP(!A+X)
```

The above example calls the macro BUMP with the first parameter consisting of the string:

```
XYZ+X
```

and not the string:

```
!A+X
```

This follows the same principle as in the previous section on macro definition. Again, it is possible to define a parameter with the text:

```
!A+X,
```

but the explanation of such a usage will be deferred until Section VII.3.A.

Parameters are substituted within text whenever the string

```
!I
```

is encountered (I=1, 2, 3,...,9) in the body of a macro. Note that any occurrence outside the body of a macro is erroneous.

```
EX 7:  !MACRO BUMP=!1 := !1 + 1 !END
       !BUMP(VAR)
```

The above example will produce the output text:

```
VAR:  := VAR + 1
```

Again, as in macro definition, if a parameter crosses one or more card boundaries and such occurrences are remembered

```
EX 8:  !MACRO EXCH =
       T := !2
       !2 := !1
       !1 := T
       !END
       !EXCH(A,B)
```

produces:

```
T := B
B := A
A := T
```

```
EX 9:  !MACRO ADD=!1 := !1 + !2!END
       !ADD(VAR1,X)
```

This produces:

```
VAR1 := VAR1 + X
```

Several precautions must be taken in parameter definition as explained by the following notes.

1. Parenthetical nesting is observed in parameter collection to allow the user to pass strings balanced with respect to parenthesis as in:

```
!BUMP(A(J))
```

This defines parameter 1 to be the string

```
A(J)
```

If nesting was not observed, then collection would end after encountering the first ')' and the parameter string would then become:

```
A(J
```

This causes a problem when one wishes to pass a single parenthesis within a parameter.

```
EX 10:  !A( ) ) or !A( ( )
```

would not correctly define a parameter string to be) or (, respectively. In the first case, the) would terminate parameter collection, and in the second, the (would be the start of a level of parametric nesting. If no corresponding) were ever encountered, the collection would continue until the end of the input text. To overcome this, parameters may be quoted.

```
EX 11:  !BUMP('XXX(XXX') or 'BUMP(')')
```

In the above cases, parameter 1 would be defined as XXX(XXX or), respectively; i.e., parenthetical nesting is ignored within quotes. As a generally accepted rule in several languages, quotes themselves are passed as double quotes within quotes.

```
EX 12:  !BUMP('ABC''DEF')
```

This defines parameter 1 to be the string

```
ABC'DEF
```

2. The same type of problem occurs when attempting to pass commas as parameter values as they are used for parameter separators. Again, to overcome this, commas must be quoted.

```
EX 13:  !BUMP('A,B')
```

This defines parameter 1 to be

```
A,B
```

If the alternate form of

```
!BUMP(A,B)
```

was instead used, then parameter 1 would be A and parameter 2 would be defined as the string B. The fact that parameter 2 might never be used in the macro body is of no consequence during parameter collection.

```
EX 14:  !MACRO ADD = !1 := !1 + !2 !END
        !ADD(A(I,J),1)
```

will result in the string

```
A(I,J) := (I,J) + 1
```

In the above case, note that the first comma (between the I and J) did not separate the parameters. Commas are only recognized as separators when they occur outside of parenthetical nesting.

3. Macros themselves are legal parameters and one such example was presented earlier.

```
EX 15:  !MACRO PROD=X*Y!END
        !MACRO SUM = !1 := !1 + !2 !END
        !SUM(Z,!PROD)
```

produces:

```
Z := Z + X*Y
```

VII.2.D Compile Time Variables

The SIMPL macro facility allows definition of compile time integer variables as follows (a BNF of all compile time definitions and statements is given in Section VII.7):

```
!INT <NAME 1>, <NAME 2>,...,<NAME N> !END
```

Integer variables can be used by themselves (much the same as normal macros) and in compile time expressions. All variables must be defined before they are used.

```
EX 16:  !INT A, B, VAR !END
```

This creates three compile time variables called A, B, and VAR, respectively. All have initial values of zero, but it is possible to initialize them to any other value at definition by replacing the string

```
<NAME>
```

with

```
<NAME>=<VALUE>
```

where <VALUE> is any integer constant.

```
EX 17:  !INT X,Y=42 !END
```

This defines two variables, X and Y, and initializes the latter to 42.

Compile time variables may appear within compile time arithmetic expressions. Such expressions are used in association with the system macros !LET, !WHILE, and !IF. Explanations of this context will be explained in the appropriate sections (VII.2.E, VII.2.F, and VII.2.G). Excluding this, they may also appear in text the same as a normal macro invocation without parameters and with the same syntax. The substituted text will then be the value of the variable at the time of its use.

```
EX 18:  !INT A=15 !END
        CALL SUB(!A)
```

The above example will produce the output string

```
     CALL SUB(15)
```

VII.2.E Compile Time Assignment Statements

The syntax for compile time assignment statement is:

```
          !LET <NAME> := <EXPR> !END
```

where <NAME> is the name of some predefined compile time variable and <EXPR> is any legal expression consisting of compile time variables, integer constants, parenthesis, and the operators listed below. The BNF description of an expression can be found in Section VII.7.

The only unary operator implemented is the unary minus (-). In addition, the following binary operations are permitted.

<OP>	Value of A <OP> B
+	arithmetic addition
-	arithmetic subtraction
*	arithmetic multiplication
/	arithmetic division *(integer with truncation)*
.AND.	1 if A and B are both nonzero, 0 otherwise
.OR.	1 if A or B is nonzero, 0 otherwise
=	1 if A equals B, 0 otherwise
<	1 if A is less than B, 0 otherwise
>	1 if A is greater than B, 0 otherwise
<=	1 if A is less than or equal to B, 0 otherwise
>=	1 if A is greater than or equal to B, 0 otherwise
<>	1 if A is not equal to B, 0 otherwise

The precedence ordering is as follows (from highest to lowest):

```
                    -    (unary minus)

                    *,/

                    +, - (binary minus)

                    =, <>, <, >, <=, >=
```

.AND.

.OR.

This precedence may be overridden by the use of parentheses.

Consider the following examples.

```
EX 19:  !LET A := B + 2 * C !END
```

This computes B + (2*C) and assigns it to A.

```
EX 20:  !LET A := (B + 2) * C !END
```

This computes (B+2) * C and assigns it to A.

```
EX 21:  !LET A := A * B .OR. C !END
```

This sets A to 1 if the computation A*B is nonzero or if C is nonzero. Otherwise, A is set to zero.

```
EX 22:  !LET A := D * (B .OR. C) !END
```

This sets A to D (D*1) if either B or C is nonzero. Otherwise, A is set to zero (D*0).

VII.2.F Conditional Text Substitution

Text may be conditionally produced as follows:

```
!IF <EXPR> !THEN <TEXT> !ELSE <TEXT> !END
```

<EXPR> is any expression as described in the preceding section on compile time assignment statements. If it evaluates to a nonzero value, then the text associated with !THEN will be processed and the !ELSE text skipped; otherwise, the !ELSE text is processed and the !THEN part skipped. Either text string may, as usual, contain other macro invocations or definitions as well as normal text. Therefore, the conditional text capabilities can be used not only to create output text but also to define and invoke macros. As in the SIMPL language, the !ELSE part is optional resulting in the alternative form:

```
!IF <EXPR> !THEN <TEXT> !END
```

In this case, if the expression evaluates to false (zero), then no text will be processed. As always, all variables used in the expression must be predefined compile time variables.

```
EX 23:  !IF A + B > 2 !THEN    STR[5] := 'X'
        !ELSE    STR[6] := 'Y'
        CALL SUB
        !END
```

If A + B is greater than 2, the string

```
          STR[5] := 'X'
```

will appear in the output text; otherwise

```
          STR[6] := 'Y'
          CALL SUB
```

```
          EX 24:  !IF I = 1 .AND. J = 2 !THEN    X := X + 1
                  !MACRO M = /+ WRITC (I, J, X) +/
                  !END
```

If I = 1 and J = 2, then the string

```
          X := X + 1
```

will be produced and the macro M defined as shown. If the condition is not true, then no text will appear and the macro M will not be defined.

```
          EX 25:  !IF I = 1 !THEN !MACRO A=TRUE !END
                  !ELSE !MACRO A=FALSE !END
                  !END
```

If I = 1, then the text of macro A will be defined as

```
          TRUE
```

else as

```
          FALSE
```

Note that !ENDs must be nested correctly and that the !END for the macro definition does not also end the !IF-!THEN-!ELSE process. This process, like any other, is always executed whenever it is encountered. This implies that an alternate form of the last example is:

```
          !MACRO A=!IF I=1 !THEN TRUE !ELSE FALSE !END !END
```

This again defines the text of macro A as either 'TRUE' or 'FALSE' depending on the value of I at the time of definition. The body of macro A does not consist of the conditional text itself and is not therefore dependent on the value of I at invocation.

VII.2.G Repetitive Text Substitution

The macro facilities allow for the rescanning of text as follows:

```
          !WHILE <EXPR> !DO <TEXT> !END
```

<EXPR> is any compile time expression as defined in Section VII.2.E. If it evaluates to nonzero, then the <TEXT> will be scanned; i.e., all output generated and any macros processed. After the scanning is complete, the expression will be

reevaluated and, if it is still nonzero, then <TEXT> will be rescanned. This process is repeated until the expression is zero after which scanning is restarted after !END assocated with the !WHILE. If the initial value of <EXPR> is zero, then the corresponding text string is never processed.

```
EX 26:  !WHILE I >= 0 !DO A(!I) := !I
        !LET I := I - 1 !END
        !END
```

If 'I' has an initial value of 5, the output will be:

```
A(5) := 5
A(4) := 4
A(3) := 3
A(2) := 2
A(1) := 1
A(0) := 0
```

and the final value of ı will be -1.

```
EX 27:  !MACRO X=X!END
        A := '!WHILE I <> 0
        !DO!X!LET I := I - 1 !END!END'
```

For this discussion, assume that I has an initial value of 3. The out
produced here will then be: put string

```
A := 'XXX'
```

The major factor to bring out in this example is that text scanning, w
expression I <> 0 is nonzero, starts with the character immediately fhile the
the !DO. Since what is wanted is a string of I X's, a macro must be pollowing
defined with the replacement text X. If the form   reviously

```
!DOX!LET ...
```

was used instead, an error would occur. No !DO would ever be found fo
corresponding !WHILE as the name collected would be !DOX. Also, as blr the
legal characters, the form: anks are

```
!DO X!LET ...
```

would yield as output

```
A := ' X X X'
```

Another point to note is that the !DO is on a separate line from the !
This is again due to the phenomenon that text collection begins immediWHILE.
the occurrence of !DO. Remembering that new lines are included in texately after
sequence: t, the

```
!MACRO X=X!END
A := '!WHILE I <> 0 !DO
X!LET I := 1!END!END'
```

would output

```
A:='
X
X
X'
```

VII.3 DELAYED MACRO INVOCATION

VII.3.A Advanced Usage

The macro facility permits advanced forms of macro nesting and delayed macro invocation. To understand these capabilities, the user must be aware that text is scanned in any of the following three instances:

1. On reading the input text, all characters are examined, including those in macro definitions and parameters.

2. On macro invocation, the macro body is rescanned.

3. On parameter substitution, the text of the parameter is rescanned.

Normally a macro, A, is invoked by

```
!A,
```

but invocation can be delayed N times (for any positive integer N) by providing N+1 !'s in front of the macro name.

Every time a string of !'s is scanned, one of the !'s is removed. Invocation does not occur until the last ! is removed. For the macro call !!!A, the macro A will then not be expanded until it is encountered for a third time. A sole exception to this rule is that no !'s are deleted from in front of parameters (e.g., !!2) when they are initially scanned from the input file.

```
EX 28:  !MACRO B=!!IF I = 1 !!THEN 'X' !!ELSE 'Y' !!END !END
```

This defines a macro B with the body:

```
!IF I = 1 !THEN 'X' !ELSE 'Y' !END
```

If this macro were used in the following text:

```
!LET I := 1 !END
A1 := !B
!LET I := 0 !END
A2 := !B
```

The output produced would be:

```
A1 := 'X'
A2 := 'Y'
```

```
EX 29:  !MACRO B = X := X + !!A !END
```

This defines macro B with the body

```
X := X + !A
```

When B is invoked, the following text will result:

```
X := X + <TEXT OF MACRO A>
```

where <TEXT OF MACRO A> is the text of macro A at the time that B is invoked, and not when B was defined. These two cases are clearly the same unless either A was redefined between the time when B was defined and the time when B was invoked, or A was not defined at all when B was defined.

```
EX 30:  !MACRO A := !!MACRO B=Y!!END X := X + !1 !END
        !A(!!B)
```

This results in the output:

```
X := X + Y
```

regardless of whether or not B had been previously defined. The only places where macro invocations (whether system or user macros) are not permitted:

1. within compile time expressions, and

2. within compile time variable definitions.

VII.3.B <u>!OPTION</u>

Macro time options may be set as follows:

```
!OPTION(<OPTION>)
```

where <OPTION> is one of the compile time options. This statement may not contain any blanks. Currently only the following are implemented:

Option	Meaning
NOLIST	Suspend listing of the input text
LIST	Resume listing of the input text
EJECT	Skip to top of next page

The choice of which one of the above options is the default is installation dependent.

VII.3.C !INCLUDE

Text from external files may be included as part of the input text by:

```
!INCLUDE(<filename>) - includes whole file
```

or

```
!INCLUDE(<filename>.<comdeckname>) - includes subset of the file
```

where <filename> is the name of the file on which the text to be included resides. No nesting of !INCLUDEs is allowed, that is, the file being "included" cannot have an !INCLUDE.

A COMDECK is a subset of the file and starts with /*+COMDECK <comdeckname>+*/ and ends with the next /*+COMDECK <comdeckname>+*/ or EOF. Note that /*+COMDECK...+*/ are treated as comments by the compiler.

VII.3.D !NULL

One macro, !NULL, is defined at initialization with a replacement text of the null string (zero characters in length). This macro may be called with or without parameters, but all parameters will be ignored. One such use might be to separate characters as in:

```
EX 31:  !IF I = 1 !THEN!NULL()ABC!END
```

IF I = 1, this produces the string ABC with no preceding blanks. The sequence:

```
!IF I = 1 !THENABC!END
```

would not work as !THENABC would not be recognized as desired. All macros undefined at invocation will be replaced by !NULL.

VII.3.E Miscellaneous

It may sometimes be desirable to use one macro to invoke a second. Assume there are five defined macros called A1, A2, A3, A4, and A5, and a compile time variable called I. One might now wish to invoke one of these macros depending on the value of I; i.e., macro A1 if I = 1, A2 if I = 2, etc. The sequence:

```
!A!I
```

will concatenate the expansion for 'A' with the expansion for I. To obtain what is wanted, the following is implemented:

```
!(A!I)
```

Here the entire string within the parentheses is expanded and used as the name of a macro to invoke. If I has the value 3, then

```
A!I
```

will expand to:

```
        A3
```

and the macro A3 will be invoked.

```
        EX 32:  !MACRO A=ABC!END !MACRO B=DEF!END
                !(!A!B)
```

This will invoke the macro ABCDEF. Macros invoked this way may be called with parameters as:

```
        EX 33:  !(!A!B) (<PARAMETER LIST>)

        EX 34:  !MACRO A=X!END    !MACRO B=Y!END
                !MACRO XY = I := !1 + !2 !END
                !(!A!B) (VAR,TEMP)
```

This will output:

```
            I := VAR + TEMP
```

VII.4 PROCESSOR CALL CARD AND OPTIONS

The QM-1 implementation may be called as any other EASY server by:

```
        MACRO, OPTION =    , INPUT =    , OUTPUT =
```

The current options implemented are:

Option	Meaning
N	Do not list input element
S	List input element
Z	List processing time

VII.5 PROCESSOR OPERATION

VII.5.A Internal Data Structure

The main storage area of the macro processor consists of one 48000 character area which is divided into the following three libraries:

1. Macro definition
2. Parameter definition
3. While loop processing

Each of these areas act as I/O buffers during execution and hold the implied corresponding information. The macro library contains macro definitions, the parameter library holds parameter values as established at macro invocation, and the while library holds the body of any WHILE block being processed. Each of these areas act as both input and output regions as information may be both stored into and retrieved from them. In addition to these I/O areas, data is also read from the standard input file and written to the standard output file. As several areas exist from which data may be obtained and into which data may be written, the macro program contains two switches, one for input and one for output, which handle all data flow within the module. By calling the corresponding routine, the next character will either be retrieved from or stored into the correct data area. Therefore, only two routines within the program keep track of any current state of data flow (see Figure VII-1).

VII.5.B Logic

The macro processor is divided into several individual routines or procedures, each handling one particular processing task. Figure VII-2 is a logic flow diagram of the complete program.

Control begins in routine 'INIT' which scans the input file character by character, reproducing it onto the output file, until a mark (!) is found. When this happens, the routine 'MAIN' is called which analyzes the string following the '!' and in turn then calls the corresponding procedure to process that string; e.g., 'MACRO DEFINITION' FOR !MACRO, 'IF PROCESSOR' for !IF, etc. If necessary, these routines can set the input or output switches to channel data flow as explained in the preceding section. During normal processing, one of these routines may legally encounter another '!' before its own processing is complete. In that case, it recursively calls the 'MAIN' routine which repeats its analysis and again calls the corresponding procedure.

The input from this second call will be obtained from the correct place since each procedure invoked alters the input switch as necessary (the same is also true for output). For example, assume that in the parameter collection for a macro, a second macro invocation without parameters is encountered. On the call to the parameter collection for the first macro routine, the output switch is set to store information into the parameter library. When the second macro is encountered, its expanded text will then be routed to this area. After the expansion is complete and the text of the remaining parameters is processed, the output switch is reset to its previous state (most likely pointing to the output file). The macro expander is then reentered to output the original macro's text, which will now appear (presumably) on the output file. Therefore, the same routine might be called twice; its output going the first time to the macro library and the second to the output file. In either case, the final goal of the data is completely transparent to any of these routines. When all macro processing is finished, control again is returned to 'INIT'.

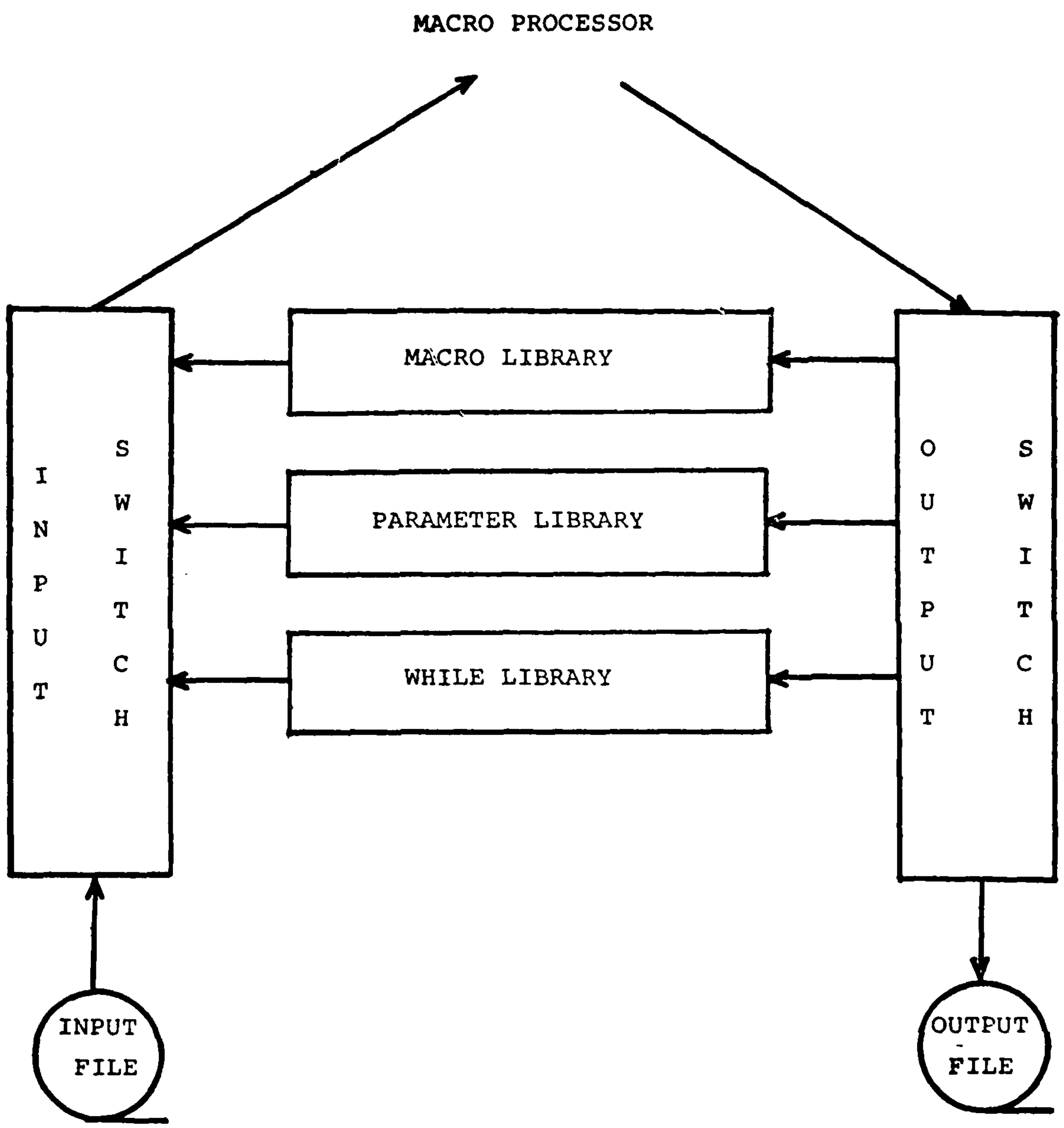


Figure VII-1. Data Flow within Macro Processor

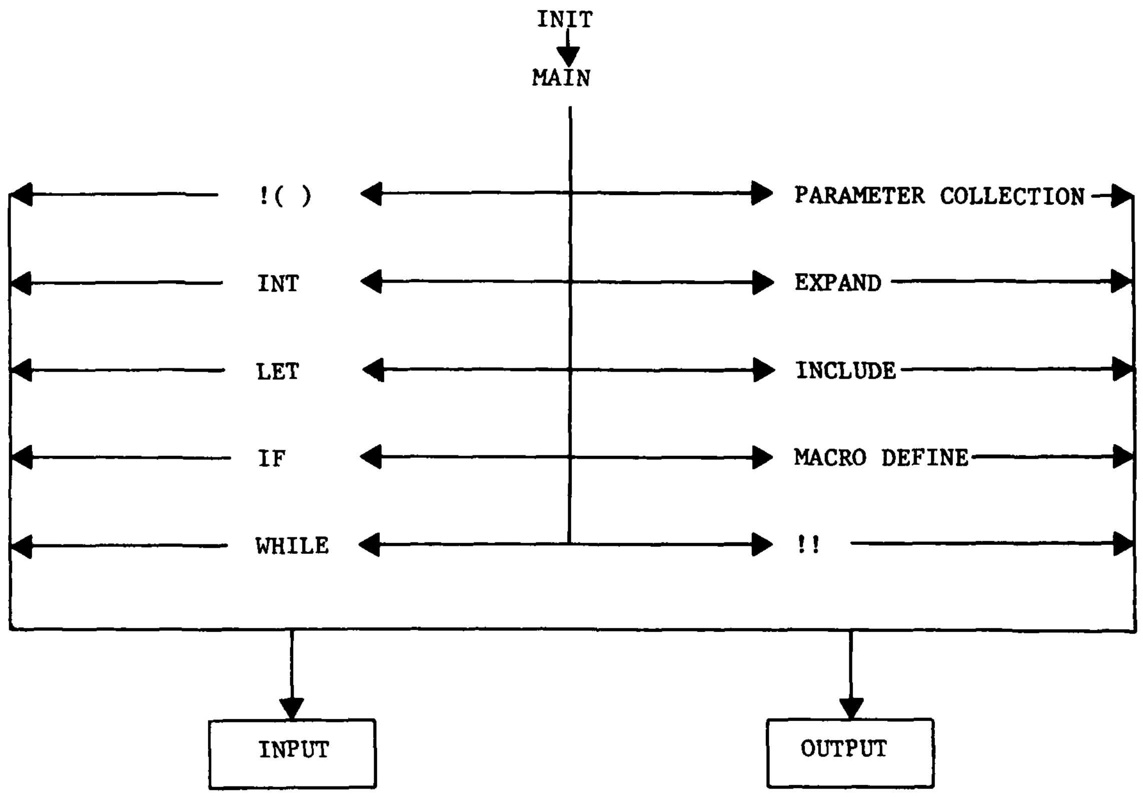


Figure VII-2. Logic Flow of the Macro Processor

VII.6 EXTENDED EXAMPLES

The following section is presented to provide some additional examples to further familiarize the user with the macro capabilities.

```
EX 35:  !MACRO ARITH=
           !!MACRO  !1=
              CLA   !!2
              !2    !!3
              STO   !!!
           !!END
        !END
```

The preceding is an example of one macro set up to define another. A usage such as:

```
!ARITH(SUM,ADD)
```

would then be equivalent to inputting the text:

```
!MACRO SUM =
   CLA   !2
   ADD   !3
   STO   !1
!END
```

The invocation of macro SUM could then be applied as any other defined macro.

```
!SUM(A,B,C)
```

would produce:

```
CLA B
ADD C
STO A
```

Notice that, since one macro may be set up to define another, the parameters of each may be determined by the number of !'s preceding them. In the above example, double marked parameters (e.g., !!1) will be associated with the innermost macro, while normal parameters (e.g., !1) will be associated with the outermost.

```
EX 36:  !MACRO CONDEF=
        !!IF I !!THEN !1 !!END
        !END
```

This macro allows for conditional parameter substitution within the body of a macro. Assume there are several debug statements a user wanted to have provided at his option. By using the above macro and setting the value of compile-time variable I at the start of the program, text could be conditionally created with the following usage:

```
!CONDEF(WRITE(VAR1,VAR2))
```

If this debug statement were desired, then I would be set to one and the output would be:

```
WRITE(VAR1,VAR2)
```

If, instead, I were set to zero, no text would occur for this macro call.

VII.7 SYNTAX AND SEMANTICS OF THE SIMPL MACRO LANGUAGE

This section contains the syntactic and semantic definition of the compile-time statements; that is, excluding macro definition and invocation, accepted by the SIMPL macro processor. It is divided into four sections:

1. Compile-time declarations
2. Compile-time statements
3. Compile-time expressions
4. Miscellaneous

In the syntactic definition, the structure {...} means that the enclosed structure is optional. Thus, {X} Y is equivalent to the BNF notation X Y | Y.

Additionally, in the syntax the problem of blanks will be ignored. At least one blank is required after an identifier (including system keywords, such as MACRO, LET, IF, etc.) whenever the next character is a letter or digit.

The term <TEXT> will not be defined, but rather its meaning is that presented in the body of the report. The problem is that a complete syntactic definition of the macro language would not be context-free.

VII.7.A <u>Compile-Time Declarations</u>

```
<compile-time declaration> ::= !INT <declaraction list> !END

<declaration list> ::= {<declaration list>,}  <declaration>

<declaration> ::= <identifier> {=<constant>}

<constant> ::= <integer> | -<integer>

<integer> ::= {<integer>} <digit>
```

All identifiers which will be used in compile-time statements must be declared before they are used. An initial value may be specified for each identifier; if none is specified, the value zero is assumed.

Compile-time variables may be invoked the same as user macros without a parameter list (even empty). In such a case, the current numeric value of the variable is substituted.

VII.7.B Compile-Time Statements

```
<compile-time assignment> ::= !LET <identifier> ::=
     <compile-time expr> !END

<compile-time if> ::= !IF <compile-time expr>
     !THEN <text> {!ELSE <text>} !END

<compile-time while> ::= !WHILE <compile-time expr>
     !DO <text> !END
```

The compile-time assignment assigns the integer value denoted by the expression to the identifier. The variable must have been previously declared in a compile-time declaration. The assignment itself is replaced by the null text.

The compile-time if is replaced by the !THEN text if the expression is nonzero (true); otherwise, by the !ELSE text (or the null text, if missing). No processing of the non-substituted text is done.

The compile-time while allows repeated processing and substitution of text. The processing and substitution continues as long as the compile-time expression evaluates to nonzero (true). No processing or substitution occurs if the expression is initially zero (false).

VII.7.C Compile-Time Expressions

```
<compile-time expr> ::= {<compile-time expr> .OR.} <logical product>

<logical product> ::= {<logical product> .AND.} <relation>

<relation> ::= {<relation> <relation op>} <simple expr>

<simple expr> ::= {<simple expr> <add op>} <term>

<term> ::= {<term> <mult op>} <factor>

<factor> ::= {<unary op>} <primary>

<primary> ::= <identifier> | <integer> | (<compile-time expr>) |
     <special parameter variable>

<relational op> ::= = | <> | > | >= | < | <=
```

<add op> ::= + | -

<mult op> ::= * | /

<unary op> ::= -

In a compile-time expression, all of the variables or identifiers must have been previously declared. The arithmetic operators are the usual addition (+), subtraction (-), multiplication (*), division (/), and unary minus (-). The relational operators produce one if true and zero if false. The operator .AND. produces zero if either operand is zero and otherwise one. The operator .OR. produces zero only if both operands are zero and otherwise one.

VII.7.D Miscellaneous

<identifier> ::= {<identifier>} <letter> | <identifier> <digit>

<letter> ::= A | B | ... | Z

<digit> ::= 0 | 1 | ... |9

<special parameter variable> ::= PAR1 | PAR2 | PAR3 | ... | PAR9

Identifiers must begin with a letter and consist of all letters and digits up to the next blank or special character. Only the first 12 characters of an identifier are significant.

VII.7.E Special Parameter Variable

Normally, compile-time expressions may consist only of compile-time variables, integer constants, parentheses, and the usual arithmetic and relational operator. Since a mark (!) is not allowed within an expression, one cannot normally use parameters within expressions within a macro body. Thus, the following example is illegal:

```
Ex:  !MACRO X = !!LET I := !1 !!END !END
```

To overcome this problem, nine variables (named PAR1, PAR2,..., PAR9) are defined during macro expansion and hold numeric values corresponding to parameters 1 to 9, respectively, in the macro invocation. Note that if a nonnumeric character, except for a minus sign, is found in a parameter body, the value of the corresponding PAR variable will be zero. (All uses of such a PAR variable are then flagged as errors.) Additionally, the null string has as a numeric value the number 0. These PAR variables may be used within compile-time expressions within macro expansions as though they were compile-time variables.

Consider the following legal example:

```
Ex.  !MACRO X =
     !!IF PAR1 = 1 !!THEN BUF(5)
          !!ELSE TEMP(2) - 1
          !!END
     !END
```

In this example, the call !X(1) produces BUF(5), while !X(0) produces TEMP(2) - 1.

The following should be noted concerning the usage of PAR variables.

1. PAR variables may only be used within an expression within a macro expansion. Note that this makes the use of !!PAR1 within a macro body incorrect.

2. PAR1, PAR2,..., PAR9 are not reserved names. Thus, these names may be used as macro names; moreover, such usage will not conflict with their use within expressions as described above. However, if they are defined as compile-time variables, the variable value will always be used in an expression, never the parameter value.

3. PAR variables may not occur on the left-hand side of a compile-assignment; i.e., their values may not be altered in the body of a macro.

APPENDIX VIII

SIMPL-Q USER'S GUIDE

(For the CDC 6700 System and Nanodata QM-1 System)

SIMPLE-Q USER'S GUIDE

(For the CDC 6700 System and Nanodata QM-1 System)

VIII.1 INTRODUCTION

At present, the SIMPL-Q compiler exists on two computer systems (CDC 6700 and Nanodata QM-1). The CDC version is a cross-compiler. That is, the CDC SIMPL-Q compiler produces EASY machine code which is transported to (usually by magnetic tape) and executed on the EASY system running on the Nanodata QM-1. Section VIII.2 describes how to use the CDC SIMPL-Q compiler. The SIMPL-Q compiler also exists on the QM-1 EASY SYSTEM. Section VIII.3 describes how to use this version.

VIII.2 SIMPL-Q ON THE CDC 6700 SYSTEM

The current SIMPL-Q cross-compiler executes under the CDC Scope 3.4 operating system. The reader is assumed to be familiar with features of the Scope 3.4 System such as: Creating Files, Job Control, Permanent Files, Intercom, Update, etc.

VIII.2.A Job Flow

Normal job flow is for the SIMPL-Q user to compile a CDC coded file containing SIMPL-Q source cards. The compiler produces a CDC binary file which is in binary (EASY machine) format. Then, using standard CDC utilities, the binary file is copied onto a CDC "stranger tape". Later, the QM-1 user will manually move the tape to the QM-1 system. Alternatively, the binary file may be shipped to the QM-1 via the 200-UT phone link using EXPORT. If there are multiple compile modules, these must first be combined into one file using WELD before copying to tape or shipping via the 200-UT phone link. See Appendix IX for a description of these support routines.

VIII.2.B SIMPL-Q Invocation Card

The current version of the compiler produces a binary file (named QM1) which contains QM1 code. The compiler uses many files. The user may change these, however, in the normal CDC execution fashion. The compiler invocation card and default file names appear below.

```
SIMPLQ,<options>,{list of files}.
```

Where options are 0 - 7 characters of compiler options as defined in VIII.2.C.

The list of files (with default names) is presented below, in order, with a description.

NOTE: All files must be mass storage files; i.e., not magnetic tape.

File Name	File Description	
<INPUT>,	Input file (SIMPL-Q source)	
<OUTPUT>,	Output file (listing)	
<QM1>,	EASY code file (binary)	
<TOK>,	Token file	internal compiler files normally not changed by user
<DATA>,	Data, initialization file	
<QUAD>,	Quad file	
<DIAG>,	Diagnostic file	
<XREF>,	Cross reference file	
<TEXT>,	Text file	

VIII.2.C Options

The compiler options are:

A - go even if severe errors are found

B - turn all debug aids off (line numbers)

C - check for array subscript (always performed)

D - check for omitted case (not supported)

F - generate attribute and cross-reference listing

I - use indirect links for forward PROC/FUNC calls

L - print QM-1 EASY code produced

M - macro pass

N - suppress printing of diagnostics

Q - print quads

R - do not generate EASY code

S - print source listing

T - activate call trace initially (not supported)

U - do not rewind QM1 file

X - abort if any diagnostic occurs

Y - activate line number trace initially (not supported)

Z - print execution time of each compiler pass

VIII.2.D How to Compile a SIMPL-Q Program

This section contains a few examples of the CDC NSWC SCOPE 3.4 control cards necessary to use the SIMPL-Q compiler both in batch and interactive mode.

VIII.2.D.1 Batch

VIII.2.D.1.1

One compile, Input from cards,
EASY code copied to tape (EASY code cannot go directly to tape under current Scope 3.4 System because of incompatibilities between FETs).

```
     Nxx,MT1.  Job card.
     ACCOUNT.  Account card.
     ATTACH,SIMPLQ,ID=N68.
     REQUEST,QMT,HI,S,IB,SV,VSN=NUxxxx,RING.  556BPI

SIMPLQ,SFL.  Listing, cross-reference, each code.
REWIND,QM1.
COPYBF,QM1,QMT,77.  Copy EASY code to tape.
7/8/9
   .
   .
   .
  SIMPL-Q source on cards
   .
   .
   .
6/7/8/9
```

VIII.2.D.1.2

Two compiles, Input from permanent files,
EASY code cataloged and copied to tape.

System Macros being used.

```
   .
   .
   .
ATTACH,SIMPLQ,ID=N68.
ATTACH,SYSMAC,ID=N68.  System Macros
ATTACH,IN1,ID=Nxx.     SIMPLQ source #1
ATTACH,IN2,ID=Nxx.     SIMPLQ source #2
REQUEST,QMT,HI,S,IB,SV,VSN=NUxxxx,RING.
REQUEST,QM1,*PF.
SIMPLQ,SFLMU,IN1.  Source, xref, EASY code, macro pass, no rewind.
SIMPLQ,SFLMU,IN2.

CATALOG,QM1. . . .  Catalog EASY code.

REWIND,QM1.
COPYBF,QM1,QMT,77.  Copy EASY code to tape.
   .
   .
   .
```

VIII.2.D.2 Intercom

VIII.2.D.2.1

Create Source, Catalog Source, Compile and Catalog Binary

```
COMMAND - EDITOR
..C S                              Create source
    enter source code
=
..S SOURCE
..CATALOG,SOURCE,ID=N68            Catalog source
..ATTACH,SIMPLQ,ID=N68             Attach compiler
..REQUEST,QM1,*PF
..SIMPLQ,SF,SOURCE,LIST            Compile, cross-reference
..CATALOG,QM1,BINARY,ID=N68        Catalog binary
..BATCH,LIST,PRINT,6868            Print listing at central site
```

VIII.2.D.2.2

Full Update, Compile Four Modules, and Ship Binaries to QM-1 via 200-UT

```
..ATTACH,OLDPL,EASYPL,ID=N68       Program library
..UPDATE,F                         Full update
..FETCH,SIMPLQ,N68                 Attach compiler
..FETCH,SYSMAC,N68                   and macrofiles
..FETCH,SYSLIB,N68
..SIMPLQ,SFUM,COMPILE,LIST  )      Compile with macro pass and
..SIMPLQ,SFUM,COMPILE,LIST  (      cross-reference, do not rewind
..SIMPLQ,SFUM,COMPILE,LIST  (      binary file (QM1)
..SIMPLQ,SFUM,COMPILE,LIST  )
..FETCH,EXPORT,N68                 Attach utility programs
..REWIND,QM1
..EXPORT,68,,,QM1                  Ship binary file to remote
..BATCH,LIST,PRINT,6868,68         Print listing at remote site
```

VIII.2.E How to Load CDC Produced Binary onto the QM-1 Under EASY

The binary for the SIMPL-Q source which has been compiled on the CDC 6700 is placed on tape or in a disk file on the 6700. Tapes must be physically carried to the QM-1 by the user. Binary disk files may be shipped to the QM-1 via the 200-UT phone link as described in the EXPORT/IMPORT User's Guide in Appendix IX. The reader is assumed to have some knowledge of EASY including how to bring it up and the use of MASTER (see EASY II System Programmer's Guide for details). Note that under EASY, all disk files used must have been preallocated using the NOVA Emulator Files Utility.

VIII.2.E.1 Binary on Magnetic Tape. The user must bring up EASY and use the MT-TO-DISK command to copy the tape to disk:

```
MT-TO-DISK,UNIT=0,FILE#=0,TO FILE=QM1,REWIND=NONE.
```

The user can then proceed to use the file, as for example, it may be edited into an existing system library by:

```
EDITLIB,LIBRARY=DDTLIB,STATUS=OLD.
REPLACE,FILE=QM1,#MODULES=1.
LIST,LEVEL=1.
<@>.
```

VIII.2.E.2 Binary Shipped to QM-1 via 200-UT Phone Link. Step 3 of the procedure for shipping binary files over the 200-UT phone link (as described in Appendix IX) is to bring up EASY and use the IMPORT command to transform the file back into its binary form and place it in a disk file under EASY.

EX:

```
     IMPORT,TO FILE=QM1
```

The user can then proceed to use the file, as for example, it may be used to create a new system library and command file:

```
EDITLIB,LIBRARY=USERLIB,STATUS=NEW.
BUILD,PASSWORD=FAILSAFE.
INCLUDE,FILE=QM1,#MODULES=1.
<@>
<@>
MAKECF,INPUT=*CDR,COMMAND FILE=JUNK.  Command file on JUNK will have
                                      template to invoke new program
LIBRARY,SEARCH 1ST=USERLIB,2ND=SYSTEM,3RD=DDTLIB.
TESTCF,COMMAND FILE=JUNK.
```

(Now user is ready to test new command.)

VIII.2.E.3 Punched Cards. A SCOPE 3.4 coded file (ASCII-64 character set) can be punched for use on the QM-1 by using standard SCOPE utilities.

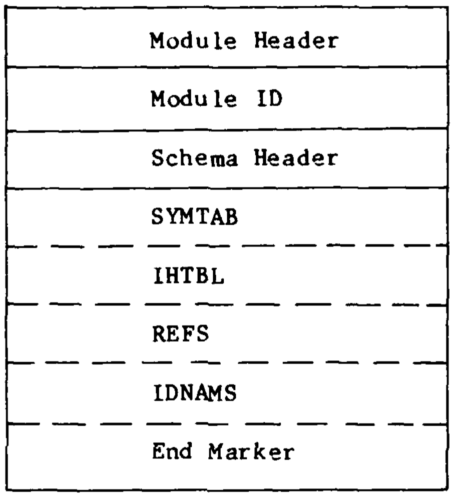
EX:

```
     ATTACH,SOURCE,ID=N68.
     COPY,SOURCE,X.
     ROUTE,X,TID=C,DC=PU,EC=029,FID=N6829.
```

VIII.3 SIMPL-Q ON THE QM-1 SYSTEM

A version of the SIMPL-Q compiler exists on the QM-1 computer running under the EASY system. The options available are the same as described for the CDC 6700 version in VIII.2.C. The default options are SFMI. The invocation of the compiler is a little different, in keeping with the use of MASTER, the EASY command interpreter. Only three files may be specified, INPUT, BINARY, and SCHEMA.

The schema file is used to facilitate symbolic debugging. If a schema file is specified, the symbol table is written to the schema file (this also forces the cross-reference option). The schema option of the SIMPL-Q compiler may be avoided by giving a filename of $ in response to SCHEMA =. The schema file, which is in the format of an EASY module, can be incorporated into a schema library by the use of EDITLIB (EASY command). The reader is directed to the _EASY II System Programmer's Guide_ for detailed information. The schema file format can be seen below.



Schema File Format

The Module Header, Module ID, and end marker are as described in the _EASY Machine Design Document_. The module name, date, and time compiled are placed in the schema module ID just as they appear in the code Module ID. The schema header contains the dimensions of the four virtual arrays, which are the body of the schema. The schema module contains no entry points.

Note that all disk files used must have been allocated using the NOVA Emulator Files Utility. To allocate the proper amount of disk space for a schema library, the number of schemas to reside on the library must be known. Allow 200 octal Nova sectors for each schema desired, plus 300 octal or so for work space.

VIII.3.A Source from Card Reader, Compile, and Test on QM-1

```
SIMPLQ,OPTIONS=SF,INPUT=*CDR,BINARY=QM1,SCHEMA=SCHEMA.  Compile program.
TEST,ENTRY=HERE,FILE=QM1.  Call ENTRY PROC here.
```

(See the use of CEXPORT/CIMPORT in the <u>EASY II System Programmer's Guide</u> for shipping source file from the CDC 6700. It is similar to the shipping of binary files.)

```
CIMPORT, TO FILE=CDR.  Copy source to file named CDR.
SIMPLQ,OPTIONS=SF,INPUT=CDR,BINARY=QM1,SCHEMA=SCHEMA.  Compile.
```

VIII.3.C Source from Card Reader, Copied to Disk, Then Compiled from Disk

```
COPY,FROM=*CDR,TO=CDR.  Copy from card reader to file named CDR.
SIMPLQ,OPTIONS=SF,INPUT=CDR,BINARY=QM1,SCHEMA=SCHEMA.  Compile.
```

APPENDIX IX

SIMPL-Q RUNTIME SUPPORT PACKAGE

SIMPL-Q RUNTIME SUPPORT PACKAGE

IX.1 EXPORT/IMPORT USER'S GUIDE

A package is available to enable the SIMPL-Q programmer using the compiler on the CDC 6700 to ship the binaries directly to the QM-1 via the 200-UT phone link, eliminating the use of tapes. On the CDC end, there is Scope 3.4, INTERCOM, and an EXPORT program. On the QM-1 end, there is the 200-UT Emulator, the EASY system, and an IMPORT Program running under EASY.

There are three steps necessary to ship a binary file from the CDC 6700 disk to the QM-1 disk under the EASY system:

1. EXPORT must be used to translate the binary file into ASCII print format and ROUTE it to the QM-1. (Currently, binary files cannot be directly sent to a 200-UT using the ROUTE command.)

2. The 200-UT Emulator must be used on the QM-1 to copy the file routed from EXPORT onto the QM-1 disk.

3. The EASY System must be brought up and the IMPORT command used to transform the file back into its binary form and place it in a disk file under the EASY system. (Step 3 must immediately follow Step 2 or the file may be destroyed by another user.)

Step 1 may be done while in batch mode, while under INTERCOM (VIII.2.D.2.2) or actually from the QM-1 itself, after bringing up the QM-1 as a 200-UT in preparation for Step 2.

The following (showing Step 1 done from the QM-1) describes the procedures to use on the QM-1 to accomplish this shipping of binary files.

The user should bring up the 200-UT Emulator on the QM-1 (see EASY System Programmer's Guide) and LOGIN under INTERCOM. All SIMPL-Q programs must be compiled and their binaries available (local or attached permanent files).

Type in the "CONTIN" command to identify the terminal as a remote batch terminal. Then, make sure the print queue is empty by using "H,O." and "Q,id." (where id is the user's 2-character INTERCOM id). One job named ididlid should be in the execute queue; others in the print queue must be eliminated. (See Queue Resident-File Control Commander in the CDC INTERCOM Reference Manual.)

Now FETCH,EXPORT,N68 and run EXPORT by typing "EXPORT,id,,,binfile." (NOTE: This step may be done prior to this time in batch mode on the CDC 6700 or under INTERCOM from another terminal.) Default binfile is QM1. The id is to be specified if the EXPORTed file is to be automatically batched to the print queue as 'ididlid'. Otherwise, type EXPORT,,,,binfile,expfile. This will leave the exported file as a local file named expfile. Default expfile name is QMSHIP.

To "copy" the export file on QM-1 disk, hit the Escape Key and type "PD" then a Carriage Return. This tells the 200-UT to "print" onto the disk instead of the printer. Enter the ON command and the file will be shipped to the QM-1. When the equipment status display comes up, the transmission is finished. Enter the OFF and LOGOUT commands.

Now that the exported file is safely on the QM-1 disk, it should immediately be brought over to EASY. Bring up the EASY system on the QM-1 and type the IMPORT command (see EASY System Programmer's Guide). The only parameter is the desired file name. The IMPORT program will restore the file to its original form and write it to the file named. (Note that under the EASY system all files used must have been preallocated.) The file is now ready to use.

APPENDIX X

SYSTEM MACROS AND SYSTEM LIBRARY

SYSTEM MACROS AND SYSTEM LIBRARY

X.1 SIMPL DOCUMENTATION MACROS

X.1.A INTRODUCTION

In order to provide a certain degree of documentation uniformity between SIMPL programs, the following set of macros has been established.

```
!MODULE
!PARAMETER
!REVISION
!ERROR
!CONTINUE
!FINISH
```

The SIMPL programmer should use the documentation macros listed above at the beginning of each compile unit. If there exists distinct modules within a compile unit, the programmer may also use the documentation macros to describe those distinct modules.

The documentation macros are made available to the SIMPL programmer by placing the card !INCLUDE(SYSMAC.DOCMAC) at the beginning of each compile unit. SYSMAC is the local file name and must be attached; the permanent file name is SYSMAC with ID=N68.

EX:

```
Job card                       )
Account card                   )
ATTACH,SIMPLQ,ID=N68           )  These control cards are
ATTACH,SYSMAC,ID=N68           >  for the CDC 6700.  Omit
.                              )  them on the QM-1.
.                              )
.                              )
7/8/9                          )
!INCLUDE(SYSMAC.DOCMAC)
.
.  SIMPL-Q SOURCE
.
```

X.1.B <u>Description of Macros</u>

X.1.B.1 <u>!MODULE(name, type, purpose, description)</u>. The macro !MODULE must be the first documentation macro to be invoked. Its purpose is to provide an overall description of the compile unit or compile unit module.

Name:	1-6 char
Type:	PROC,STRING,INT,CHAR
Purpose:	1-40 char
Description:	1-___ char (no commas or parenthesis may be part of the description; do not write past col 72)

X.1.B.2 !PARAMETER(name, type, comment). !PARAMETER is an optional macro used to describe parameters.

Name:	1-6 char
Type:	STRING,INT,CHAR,FILE
Comment:	1-40 char (no commas or parenthesis may be part of the comment)

NOTE: Do not leave spaces between commas/parenthesis and call line parameters.

EXAMPLE: !PARAMETER(II,INT,COUNTER)

X.1.B.3 !REVISION(date, programmer, reason). The !REVISION macro is mandatory and must follow all !PARAMETER macros if any !PARAMETER macros are invoked.

Date:	mm/dd/yy
Programmer:	1-9 char
Reason:	1-40 char (no commas or parenthesis may be part of the comment)

NOTE: Do not leave spaces between commas/parenthesis and call line parameters.

EXAMPLE: !REVISION(01/25/76,J DOE,INITIAL VERSION)

X.1.B.4 !ERROR(message). The !ERROR macro is used to describe any error codes or error messages used by the compile unit. The !ERROR macro is optional; however, if used, an !ERROR macro must follow all !REVISION macros. The message parameter of the !ERROR macro may be from 1-40 characters (no commas or parenthesis allowed).

X.1.B.5 !CONTINUE(message). The !CONTINUE macro is used to extend the comment, reason, and message parameters of the !PARAMETER, !REVISION, and !ERROR macros, respectively. The message parameter of the !CONTINUE macro may be 1-40 characters (no commas or parenthesis allowed).

X.1.B.6 !FINISH. The !FINISH macro is manadatory, has no parameters, and must be the last documentation macro invoked in any sequence of documentation macros.

X.1.B.7 PRINT CONTROL. A macro variable $PRMAC controls whether or not the included file source is listed by the SIMPL-Q compiler. If $PRMAC is set to 1, then the included file is listed. If $PRMAC is set to 0, the included macro file is not listed.

EXAMPLE INPUT:

```
!INT $PRMAC=0 !END  (no listing of included file)
!INCLUDE(SYSMAC.DOCMAC)
!MODULE(MAIN,PROC,TEST PROGRAM,MACRO DEMONSTRATION)
!PARAMETER(II,STRING,THE CURRENT CONTENTS)
!CONTINUE(OF THE PRINT BUFFER)
```

```
!PARAMETER(JJ,INT,CURRENT PRINT BUFFER LENGTH)
!REVISION(01/02/74,DAWSON,INITIAL VERSION)
!REVISION(01/05/75,JONES,OPTIMIZATION)
!ERROR(CODE 001 IMPLIES BAD DATA)
!FINISH
!INCLUDE(SYSLIB.SYSET)
  .
  .
  .
!INCLUDE(SYSLIB.CONSET)
/+EJECT+/
  { SIMPL-Q source code
```

X.1.C Sample Output

```
     SIMPL-Q 1.70 CDC version 1.8  13.33.24.  02/10/76
82        /* MODULE MAIN {TEST PROGRAM} */

82        /*********************************************************************
82        * MODULE NAME = MAIN
82        * MODULE TYPE = PROC
82        * MODULE PURPOSE = TEST PROGRAM
82        *
82        *
82        * MODULE DESCRIPTION
82        ------------------
82        MACRO DEMONSTRATION
82        ********************************************************************/

83        /********************************************************************/
83        /* PARAMETERS                                                      */
83        /* NUM   NAME   TYPE   COMMENT                                     */
83        /* ---   ----   ----   -------                                     */
83        /*  1     II                                                       */
83                       STRING
83                              THE CURRENT CONTENTS */
84        /*                    OF THE PRINT BUFFER */
85        /*  2     JJ
85                       INT
85                              CURRENT PRINT BUFFER LENGTH */
86        /********************************************************************/

86        /********************************************************************/
86        /* REVISIONS                                                       */
86        /*                                                                 */
86        /*   DATE       PROGRAMMER  REASON                                 */
86        /*   ----       ----------  ------                                 */
86        /*  01/02/74   DAWSON
86                                  INITIAL VERSION */
87        /*  01/05/75   JONES
87                                  OPTIMIZATION */
88        /********************************************************************/
```

```
/*******************************************************************/
/* ERRORS                                                          */
/*                                                                 */
/*                      ERROR:  CODE 001 IMPLIES BAD DATA */
/*******************************************************************/
```

X.2 SYSLIB

SYSLIB is the SIMPL-Q system Macro library. The library is divided into different sections called COMDECKS. Each COMDECK has a unique name and falls into one of the following categories.

1. SETUP - provides linkage to external SIMPL-Q routines and data items.

2. CODE - SIMPL-Q text.

3. GLOBALS - SIMPL-Q system globals.

In order to utilize a particular COMDECK, the user must first include the system documentation macros (via the system macros file) and then include the system library file specifying the desired COMDECK. For example,

```
!INCLUDE(SYSMAC.DOCMAC)
     .
     .
     .
!INCLUDE(SYSLIB.comdeck name 1)
     .
     .
     .
!INCLUDE(SYSLIB.comdeck name 2)
     .
     .
     .
```

The following COMDECKS are currently in the system library file.

Name	Type	Description
CONSET	SETUP	Provides linkage to the SIMPL-Q conversion routines.
KBDSET	SETUP	Provides linkage to the SIMPL-Q keyboard read routines.
CRTSET	SETUP	Provides linkage to CRT support routines and data items.
FPSET	SETUP	Provides linkage to EASY Floating Point support routines.
SYSTXT	CODE	Contains source code for system routines.

Name	Type	Description
TRDGLB	GLOBALS	Globals needed to interface with TRIDENT EMULATION.
MASTDEF	GLOBALS	Configuration constants for MASTER.
SYSET	SETUP	Linkage to EASY system routines.
ENTSET	SETUP	Linkage to DDT file handlers.
MTSET	SETUP	Linkage to mag tape routines.
CIOSET	SETUP	Linkage to system circular Input/Output routines.

A detailed description of the routines, data, etc. in each COMDECK is available from K74. Also, the EASY II System Programmer's Guide contains a discussion.

DISTRIBUTION

USC/Information Sciences Institute
4676 Admirality Way
Marina Del Ray, CA 90291
ATTN: Lou Gallenson

Defense & Space Systems Group of TRW Inc.
One Space Park
Redondo Beach, CA 90278
ATTN: David Bixler
Herb Wangenheim

John S. Hale
P.O. Box D5
Stony Brook, NY 11790

Dr. John Tartar
Department of Computer Science
University of Alberta
Edmonton Alberta
Canada T6G 2E1

Jon Humphry
Hughes Aircraft Company
Bldg. 12 M/S V107
Culver City, CA 90230

Commander
Naval Ocean Systems Center
271 Catalina Boulevard
San Diego, CA 92152
ATTN: Code 5200
Russ Eyers

Commanding Officer
U.S. Army Harry Diamond Laboratories
2800 Powder Mill Road
Adelphi, MD 20783
ATTN: Branch 520
Rick Johnson

Defense Technical Information Center
Cameron Station
Alexandria, VA 22314 (10)

U.S. Naval Electronic Systems Command
Washington, D.C. 20360
ATTN: John Machado
(Code 330)

DISTRIBUTION (Cont'd)

Robert V. Hanzlian
224 Courtland St.
Buffalo, NY 14205

Director
U.S. Army TRADOC System Analysis
Activity
White Sands Missile Range, NM 88002
ATTN: ATAA-SL (Technical Library)

Computer Science Department
University of Maryland
College Park, MD 20742
ATTN: Dr. Yaohan Chu
Dr. Victor Basili

Paul A. Boudreaux
University of Southwestern Louisiana
USL Box 4-4330
Lafayette, LA 70504

DIT-MCO International
5612 Brighton Terrace
Kansas City, MO 64130
ATTN: J. L. Herbsman

Boeing Commercial Airplane Company
P.O. Box 3707
Seattle, WA 98124
ATTN: Keith Mitchell, M.S. 9R-08
Flight Control Lab

Hughes Aircraft Company
P.O. Box 92426
Los Angeles, CA 90009
ATTN: Steve Jackson, M.S. R1/B218

Intermetrics, Inc.
5392 Bolsa Avenue
Huntington Beach, CA 92647
ATTN: Marjorie Kirchoff

McDonnell Douglas Astronautics Co.
5301 Bolsa Avenue
Huntington Beach, CA 92647
ATTN: S. Harris Dalrymple, M.S. A3-236-10-1

Robins Air Force Base
Warner Robins, GA 31098
ATTN: John Douglas
ACL/MMECV

DISTRIBUTION (Cont'd)

USAF RADC
Griffiss AFB, NY 13441
ATTN: Armand Vito, ISCA

Tinker Air Logistics Center
Tinker AFB
Oklahoma City, OK 73145
ATTN: Gary M. Parish, MMECO

Nanodata Computer Corporation
One Computer Park
Buffalo, NY 14203
ATTN: Mike Senft
Bill Robertson

System Development Corporation
601 Caroline Street
Fredericksburg, VA 22401
ATTN: Paul Tiffany

Douglas Martindale
OC-AOC/MMECO
Tinker AFB
Oklahoma City, OK 73145

Local:	
K33	(40)
K10	
K20	
K40	
K50	

ILME